

Datalog Rewriting for Guarded TGDs

Michael Benedikt¹, Maxime Buron², Stefano Germano¹, Kevin Kappelmann³ and Boris Motik¹

¹Oxford University, Parks Road, Oxford, OX1 3QD, United Kingdom

²LIRMM, Inria, Univ. of Montpellier, Montpellier, France

³Technical University of Munich, Boltzmannstraße 3, Garching, 85748, Germany

Abstract

We deal with the problem of fact entailment with respect to a database and a set of integrity constraints, focusing on the case of Guarded tuple-generating dependencies (GTGDs). The original approach to the problem in the literature is via forward reasoning or “chasing”, where one completes the input database by adding fresh elements and facts. This completion process may be infinite, but in the case of GTGDs it is known that one can compute a point where the chase can be cut off without missing any base facts. Another approach is by forming an automaton and checking it for emptiness. Neither of these approaches scales to large input datasets. An alternative approach is to *rewrite the constraints into Datalog*: the Datalog rewriting can be generated in advance of any dataset and will produce the same base facts as the original constraints. It is known that Datalog rewritings always exist. But to our knowledge the approach has never been implemented. In this work we overview effective algorithms to Datalog rewriting of GTGDs. This presents work that will appear in VLDB 2022.

Keywords

Chase, Datalog, Fact Entailment, Guarded TGDs, Resolution, Rewriting

1. Introduction


Reasoning with dependencies has played a large role in database theory. Dependencies can be used to describe semantic restrictions on sources, mapping rules between datasources and virtual data objects in data integration, and semantic rules on virtual data that allow new data items to be derived. A fundamental computation problem associated with dependencies is *query answering*: given a query Q , as an existentially quantified conjunction of atoms (a *conjunctive query*), a collection of facts I , and a set of dependencies Σ , find all the answers to Q that can be derived from I via reasoning with Σ . For general classes of dependencies, such as tuple-generating dependencies (TGDs) and equality-generating dependencies, query answering is undecidable. Thus, much early work focused on dependencies for which query answering is decidable, with two families being those with *terminating chase*: 1) those for which *forward-reasoning* by inferring all new facts from I will terminate in a finite number of steps – weakly-acyclic dependencies are perhaps the best-known class with terminating class – and 2) those for which *backwards reasoning* terminates – these are often known as *first-order*

Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry, September 05, 2022, Genova - Nervi, Italy

✉ michael.benedikt@cs.ox.ac.uk (M. Benedikt); maxime.buron@inria.fr (M. Buron); stefano.germano@cs.ox.ac.uk (S. Germano); kevin.kappelmann@tum.de (K. Kappelmann); boris.motik@cs.ox.ac.uk (B. Motik)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

rewritable classes. Another attractive class of dependencies that emerged in the last decade are guarded tuple-generating dependencies (GTGDs). Guarded TGDs allow to express simple referential constraints on source instances, common mapping rules between sources and targets, and target constraints in common description logic and ontology languages. Query-answering for GTGDs has long been known to be decidable [1]. They are also interesting and challenging in that they do not in general have a terminating chase and are not first-order rewritable.

There are several ways to derive the decidability of GTGD query-answering. One can argue that GTGDs have the tree-like model property – it suffices to look at structures that can be coded by trees – and then argue via reduction to Monadic Second Order Logic over trees, which was shown decidable by Rabin [2]. A refinement is to argue for the tree-like model property, directly generate an automaton that accepts codes of tree-like models, and test non-emptiness of the automaton [3]. Another variant is to create a Tableau calculus that tries to build a tree-like model, using a form of *blocking* to ensure termination. This approach is widely-applied in the setting of description logics [4] and was later lifted to the setting of guarded logics [5]. The first approach is infeasible in practice. The second suffers from two problems: building the automaton is extremely expensive and the non-emptiness test is both complex and expensive. The third approach is more plausible, but it requires a huge amount of non-determinism in choosing what tableau rules to fire in addition to the complexity of the blocking condition.

An alternative approach is based on *Datalog rewriting*: starting from Σ alone, we create a Datalog program $\text{rew}(\Sigma)$ which produces the same base facts as Σ for any dataset I . Datalog rewriting has its origins in work of Marnette [6], but has been extended to wider classes of constraints [7, 8] and refined to get more information on the program [9]. Since Σ and $\text{rew}(\Sigma)$ entail the same base facts on I , we can answer any *existential-free* conjunctive query (i.e. queries where all variables are answer variables). The restriction to existential-free queries is technical: existentially quantified variables in a query can be matched to objects introduced by existential quantification, and these are not preserved in a Datalog rewriting. However, practical queries are typically existential-free since all query variables are usually answer variables. The rewriting approach has the advantage of being scalable in the data. Moreover, $\text{rew}(\Sigma)$ can be run using optimized Datalog engines [10, 11]. While the approach has been implemented in the setting of description logics [12, 13], concrete algorithms in the setting of GTGDs have not yet appeared.

The goal of this work is to give an account of Datalog rewriting for GTGDs. We will explain the relationship to the tree-like model approach and provide an abstract framework, giving sufficient conditions that guarantee completeness of a Datalog rewriting algorithm. After providing an initial algorithm that instantiates the framework, we show how the algorithm can be optimized. We implemented the algorithms and performed an experimental evaluation, showing that they are competitive and useful in practice. The GitHub repository [14] contains the associated codebase and experiments.

2. Related Work

Decidability of query answering for GTGDs was shown in [1], based on cutting off the chase. This technique has been extended to wider classes (e.g. frontier-guarded TGDs [7]).

Answering queries via rewriting originated within description logics. The original emphasis

was on queries and constraints where there is a rewriting as a union of conjunctive queries (UCQs), as in the DL-Lite family [15]. Datalog rewriting was extensively explored in the context of ontologies in [12], with an implementation in [10]. In the context of TGDs, rewritings were first considered in the context of inclusion and key dependencies [16], where again one can obtain a UCQ rewriting. Datalog-rewritability for certain answers of conjunctive queries with respect to GTGDs was shown first in [6]. This was extended to frontier-guarded TGDs in [9] and nearly frontier-guarded and nearly guarded TGDs [8]. While Datalog rewriting has been implemented for separable and weakly separable TGDs [17], it has not been implemented for GTGDs to our knowledge.

Our work relies on connections between query rewriting and specialized versions of the chase, dating back to work on answering queries over data sources with access patterns [18]. It was later explored for GTGDs and disjunctive GTGDs in [19]. This paper presents work that is to appear in a longer version in VLDB.

3. Preliminaries

We assume the usual notion of instance and formulas, based on infinite sets vars of variables and Vals of values. We assume that Vals contains every element that will occur in an instance. For α a formula or a set thereof, $\text{consts}(\alpha)$ and $\text{vars}(\alpha)$ are the sets of constants and free variables, respectively, in α .

Dependencies. A *tuple generating dependency* (TGD) is a first-order formula of the form: $\forall \vec{x}[\beta(\vec{x}) \rightarrow \exists \vec{y} \eta(\vec{x}, \vec{y})]$ where β and η are conjunctions of atoms. β is referred to as the *body* and η as the *head*. A TGD is *full* if the head contains no existential quantifiers. A TGD is in *head-normal form* if it is full and its head contains exactly one atom, or it is non-full and each head atom contains at least one existentially quantified variable. Each TGD can be easily transformed to an equivalent set of TGDs in head-normal form. A full TGD in head-normal form is a *Datalog rule*, and a *Datalog program* is a finite set of Datalog rules. The *head-width* of a TGD τ ($\text{hwidth}(\tau)$) is the number of variables in the head; this is extended to sets of TGDs by taking the maxima over all TGDs. A *Guarded TGD* (GTGD) is one where at least one atom in β contains all the variables of β .

Fact Entailment. Given a set of TGDs Σ and a set of facts I , the *ground closure of I under Σ* is the set of facts F using only constants from I (so-called *base facts*) that can be derived from I using Σ . The *fact entailment problem* is to decide whether a given base fact F is in the ground closure of I under Σ . We are interested in deciding fact entailment for GTGDs.

The Chase. Fact entailment for TGDs is semi-decidable, and many variants of the *chase* can be used to define a (possibly infinite) set of facts that contains precisely all base facts entailed by an instance and a set of TGDs. By drawing inspiration from techniques for reasoning with guarded logics [20, 21] and referential database constraints [22], fact entailment for GTGDs can be decided by using a variant of the chase that works on tree-like structures. Specifically, a *chase tree* T consists of a directed tree, one tree vertex that is said to be *recently updated*, and a function mapping each vertex v in the tree to a finite set of facts $T(v)$. A chase tree T can be transformed to another chase tree T' in the following two ways.

- One can apply a *chase step* with a GTGD $\tau = \forall \vec{x}[\beta \rightarrow \exists \vec{y} \eta]$ in head-normal form. The precondition is that there exist a vertex v in T and a substitution σ with domain \vec{x} such that $\sigma(\beta) \subseteq T(v)$. The result of the chase step is obtained as follows.
 - If τ is full (and thus η is a single atom), then chase tree T' is obtained from T by making v recently updated in T' and setting $T'(v) = T(v) \cup \{\sigma(\eta)\}$.
 - If τ is not full, then σ is extended to a substitution σ' that maps each variable in \vec{y} to a labeled null not occurring in T . The chase tree T' is then obtained from T by introducing a fresh child v' of v , making v' recently updated in T' , and setting $T'(v') = \sigma'(\eta) \cup \{F \in T(v) \mid F \text{ is } \Sigma\text{-guarded by } \sigma'(\eta)\}$, where F is Σ -guarded by some set of facts S if $\text{const}(F) \subseteq \text{const}(F') \cup \text{const}(\Sigma)$ for some $F' \in S$.
- One can apply a *propagation step* from a vertex v to a vertex v' in T . Chase tree T' is obtained from T by making v' recently updated in T' and setting $T'(v') = T(v') \cup S$ for some nonempty set S satisfying $S \subseteq \{F \in T(v) \mid F \text{ is } \Sigma\text{-guarded by } T(v')\}$.

A *tree-like chase sequence* for an instance I and a finite set of GTGDs Σ in head-normal form is a finite sequence of chase trees T_0, \dots, T_n such that T_0 contains exactly one *root vertex* r that is recently updated in T_0 and $T_0(r) = I$, and each T_i with $0 < i \leq n$ is obtained from T_{i-1} by a chase step with some $\tau \in \Sigma$ or a propagation step. For each vertex v in T_n and each fact $F \in T_n(v)$, this sequence is a *tree-like chase proof of F from I and Σ* . It is well known that $I, \Sigma \models F$ if and only if there exists a tree-like chase proof of F from I and Σ (e.g. [1]).

Rewriting. A *Datalog rewriting* of a finite set of TGDs Σ is a Datalog program $\text{rew}(\Sigma)$ such that $I, \Sigma \models F$ if and only if $I, \text{rew}(\Sigma) \models F$ for every instance I and base fact F . If Σ contains GTGDs only, then a Datalog rewriting $\text{rew}(\Sigma)$ is guaranteed to exist (which is not the case for general TGDs). Thus, we can reduce fact entailment for GTGDs to Datalog reasoning, which can be solved using highly optimized Datalog techniques.

4. The Chase and Datalog Rewriting

Our objective is to develop algorithms that compute the rewriting of GTGDs. Intuitively, each algorithm will derive Datalog rules that summarize sequences of steps in a tree-like chase proof. Instead of introducing a child vertex v' by using a chase step with a non-full GTGD at vertex v , performing some inferences in v' , and then propagating a derived fact F back from v' to v , these “shortcuts” will derive F in one step without having to introduce v' . The main question is how to keep the number of derived rules low while still being able to derive all sequences necessary for completeness. A key observation is that instead of considering arbitrary chase proofs, we can restrict our attention to chase proofs that are *one-pass* according to Definition 4.1 below.

Definition 4.1. A *tree-like chase sequence* T_0, \dots, T_n for an instance I and a finite set of GTGDs Σ in head-normal form is *one-pass* if, for each $0 < i \leq n$, chase tree T_i is obtained by applying one of these two steps to the recently updated vertex v of T_{i-1} :

- a *propagation step* copying exactly one fact from v to its parent, or

- a chase step with a GTGD from Σ provided that no propagation step from v to the parent of v is applicable.

Thus, each step in a one-pass tree-like chase sequence is applied to a “focused” vertex; steps with non-full TGDs move the “focus” from parent to child, and propagation steps move the “focus” in the opposite direction. Moreover, once a child-to-parent propagation takes place, the child cannot be revisited in further steps. Theorem 4.2 states a key property about chase proofs for GTGDs: whenever a proof exists, there exists a one-pass proof too.

Theorem 4.2. *For every instance I , each finite set of GTGDs Σ in head-normal form, and each base fact F such that $I, \Sigma \models F$, there exists a one-pass tree-like chase proof of F from I and Σ .*

One-pass chase proofs are interesting because they can be decomposed into *loops*: a subsequence T_i, \dots, T_j of chase steps that move the “focus” from a parent to a child vertex, perform a series of inferences in the child and its descendants, and finally propagate one fact back to the parent. If non-full TGDs are applied to the child, then the loop can be recursively decomposed into further loops at the child. The properties of the one-pass chase ensure that each loop is finished as soon as a fact is derived in the child that can be propagated to its parent, and that the vertices introduced in the loop are not revisited at any later point in the proof. This way, each loop at vertex v can be seen as taking the set $T_i(v)$ as input and producing the output fact F that is added to $T_j(v)$. This leads us to the following idea: for each loop with the input set of facts $T_i(v)$, a rewriting should contain a “shortcut” Datalog rule that derives the loop’s output.

These ideas are formalized in Proposition 4.3, which will provide us with a correctness criterion for our algorithms.

Proposition 4.3. *A Datalog program Σ' is a rewriting of a finite set of GTGDs Σ in head-normal form if*

- Σ' is a logical consequence of Σ ,
- each Datalog rule of Σ is a logical consequence of Σ' , and
- for each instance I , each one-pass tree-like chase sequence T_0, \dots, T_n for I and Σ , and each loop T_i, \dots, T_j at the root vertex r with output fact F , there exist a Datalog rule $\beta \rightarrow H \in \Sigma'$ and a substitution σ such that $\sigma(\beta) \subseteq T_i(r)$ and $\sigma(H) = F$.

Intuitively, the first condition ensures soundness: rewriting Σ' should not derive more facts than Σ . The second condition ensures that Σ' can mimic direct applications of Datalog rules from Σ at the root vertex r . The third condition ensures that Σ' can reproduce the output of each loop at vertex r using a “shortcut” Datalog rule.

5. Rewriting Algorithms

We now consider ways to produce Datalog rules that “shortcut” all necessary steps in the chase. These are motivated and proven correct using Proposition 4.3. Each method is presented as a

set of inference rules that derive new rules (either full or non-full) from existing ones. We will then close the original set of rules under this inference process. Since this inference process is closely related to resolution in classical theorem proving [23], we sometimes refer to the output of a step as a *resolvent*.

In this short paper, we omit details of the algorithm that uses the inference rules to perform the closure: see [24]. The algorithm maintains sets of rules that have not yet been processed, and performs an iteration. At each step it performs redundancy elimination, important not only for efficiency but also for termination, and normalization (e.g. conversion to head normal form, choosing canonical names for variables).

The FullDR Algorithm: Creating Datalog Rules Directly. Our first algorithm will create only full (aka Datalog) rules. Similar algorithms have appeared in the prior literature for related TGD classes [8, 18]. The main interest here is that this algorithm will serve as a baseline.

Definition 5.1. *The Full Datalog Rewriting inference rule FullDR can be applied in two ways, depending on the types of TGDs it takes.*

- The (COMPOSE) variant of the FullDR inference rule takes full TGDs

$$\tau = \forall \vec{x}[\beta \rightarrow A] \quad \text{and} \quad \tau' = \forall \vec{z}[A' \wedge \beta' \rightarrow H']$$

and a substitution θ such that

- $\theta(A) = \theta(A')$,
 - $\text{dom}(\theta) = \vec{x} \cup \vec{z}$, and
 - $\text{rng}(\theta) \subseteq \vec{w} \cup \text{consts}(\tau) \cup \text{consts}(\tau')$, where \vec{w} is a vector of $\text{hwidth}(\Sigma) + |\text{consts}(\Sigma)|$ variables different from $\vec{x} \cup \vec{z}$,
- and it derives $\theta(\beta) \wedge \theta(\beta') \rightarrow \theta(H')$.

- The (PROPAGATE) variant of the FullDR inference rule takes TGDs

$$\tau = \forall \vec{x}[\beta \rightarrow \exists \vec{y} \eta \wedge A_1 \wedge \dots \wedge A_n] \quad \text{and} \quad \tau' = \forall \vec{z}[A'_1 \wedge \dots \wedge A'_n \wedge \beta' \rightarrow H']$$

and a substitution θ such that

- $\theta(A_i) = \theta(A'_i)$ for each i with $1 \leq i \leq n$,
 - $\text{dom}(\theta) = \vec{x} \cup \vec{z}$,
 - $\text{rng}(\theta) \subseteq \vec{w} \cup \vec{y} \cup \text{consts}(\tau) \cup \text{consts}(\tau')$, where \vec{w} is a vector of $\text{hwidth}(\Sigma) + |\text{consts}(\Sigma)|$ variables different from $\vec{x} \cup \vec{y} \cup \vec{z}$,
 - $\theta(\vec{x}) \cap \vec{y} = \emptyset$, and
 - $\text{vars}(\theta(\beta')) \cap \vec{y} = \emptyset$ and $\text{vars}(\theta(H')) \cap \vec{y} = \emptyset$,
- and it derives $\theta(\beta) \wedge \theta(\beta') \rightarrow \theta(H')$.

The rough idea is that the (PROPAGATE) variant simulates generation of a fact via propagation from a child to its parent in the chase: We generate a child using one of the original rules and the child gains a fact F guarded by the parent using a derived rule. Then (PROPAGATE) generates a rule that adds F directly. The (COMPOSE) variant simulates transitive fact derivations inside

a node: at a node v , one derived Datalog rule will generate a fact F_1 , and then a second derived Datalog rule will use the facts of v unioned with F_1 to generate an additional fact F_2 . The (COMPOSE) variant will generate F_2 directly from v .

The FullDR algorithm has several obvious weak points. First, it considers all possible ways to compose Datalog rules as long as this produces a rule with at most $\text{hwidth}(\Sigma) + |\text{consts}(\Sigma)|$ variables. Second, it is not clear how one efficiently obtains the atoms A_1, \dots, A_n and A'_1, \dots, A'_n participating in the (PROPAGATE) variant. Third, the number of substitutions θ in the (COMPOSE) and (PROPAGATE) variants of the FullDR inference rule can be very large.

Nevertheless, we implemented FullDR using subsumption and indexing techniques used for the other algorithms. Unsurprisingly, we did not find it competitive in our experiments.

The Existential-Based Rewriting. We now consider an algorithm that generates both non-full and Datalog rules inductively. Before defining the algorithm formally, we introduce a refined notion of unification.

Definition 5.2. For X a set of variables, an X -MGU θ of atoms A_1, \dots, A_n and B_1, \dots, B_n is a most general unifier (MGU) of A_1, \dots, A_n and B_1, \dots, B_n with the additional requirement that $\theta(x) = x$ for each $x \in X$.

It is straightforward to see that an X -MGU is unique up to the renaming of variables not contained in X , and that it can be computed as usual while treating variables in X as if they were constants. We are now ready to formalize the ExbDR algorithm.

Definition 5.3. The Existential-Based Datalog Rewriting inference rule ExbDR takes GTGDs

$$\tau = \forall \vec{x} [\beta \rightarrow \exists \vec{y} \eta \wedge A_1 \wedge \dots \wedge A_n] \quad \text{and} \quad \tau' = \forall \vec{z} [A'_1 \wedge \dots \wedge A'_n \wedge \beta' \rightarrow H']$$

with $n \geq 1$, and a \vec{y} -MGU θ of A_1, \dots, A_n and A'_1, \dots, A'_n such that

$$\theta(\vec{x}) \cap \vec{y} = \emptyset \quad \text{and} \quad \text{vars}(\theta(\beta')) \cap \vec{y} = \emptyset,$$

and it derives $\theta(\beta) \wedge \theta(\beta') \rightarrow \exists \vec{y} \theta(\eta) \wedge \theta(A_1) \wedge \dots \wedge \theta(A_n) \wedge \theta(H')$.

The intuition behind ExbDR is that it mimics two kinds of evolutions in a one-pass chase proof. One kind starts with a parent, produces a child node and then evolves it (via some loops rooted at the child). This evolution is mimicked by producing a non-full TGD that directly produces the updated child from the parent. The evolution of this child may eventually include a fact that is guarded by the parent, and hence could be propagated to the parent. The ExbDR rule also captures this propagation by again deriving a rule with existentials that directly produces the updated child from the parent; but when this rule is converted into head normal form, it will break down into a non-full and a full rule, the latter of which mimics the propagation step.

Example 5.4. Consider the set of GTGDs $\Sigma = \{\tau_1, \dots, \tau_4\}$, where

$$\begin{aligned} \tau_1 &= R(x_1) \rightarrow \exists y_1, y_2 T(x_1, y_1, y_2), & \tau_2 &= T(x_1, x_2, x_3) \rightarrow \exists y U(x_1, x_2, y), \\ \tau_3 &= U(x_1, x_2, x_3) \rightarrow V(x_1, x_2), & \tau_4 &= T(x_1, x_2, x_3) \wedge V(x_1, x_2) \wedge S(x_1) \rightarrow M(x_1), \end{aligned}$$

We can first apply the ExbDR rule to τ_2 and τ_3 to obtain the Datalog rule $\tau_5 = T(x_1, x_2, x_3) \rightarrow V(x_1, x_2)$. We can then take τ_1 and τ_5 to derive the non-full $\tau_6 = R(x_1) \rightarrow \exists y_1, y_2 (T(x_1, y_1, y_2) \wedge V(x_1, y_1))$. Finally, we can take τ_6 and τ_4 to obtain $\tau_7 = R(x_1) \wedge S(x_1) \rightarrow M(x_1)$. No further ExbDR steps are possible and the set $\text{rew}(\Sigma) = \{\tau_3, \tau_4, \tau_5, \tau_7\}$ is returned.

Using Skolemization. The ExbDR algorithm exhibits two drawbacks. First, each application of the ExbDR inference combines the head atoms of two rules, so the rule heads can get very long. Second, each inference requires matching a subset of body atoms of τ' to a subset of the head atoms of τ . This can be costly, particularly when rule heads are long.

We would ideally derive GTGDs with a single head atom and unify just one body atom of τ' with the head atom of τ , but this does not seem possible if we stick to manipulating GTGDs: doing so would require us to refer to the same existentially quantified object in different GTGDs. However, this can be achieved by replacing existentially quantified variables by Skolem terms, which in turn gives rise to the SkDR algorithm from Definition 5.5.

Definition 5.5. *The Skolem Datalog Rewriting inference rule SkDR takes two Skolemized single-headed rules*

$$\tau = \beta \rightarrow H \quad \text{and} \quad \tau' = A' \wedge \beta' \rightarrow H'$$

such that

- β is Skolem-free and H contains a Skolem symbol, and
- A' contains a Skolem symbol, or τ' is Skolem-free and A' contains all variables of τ' ,

and, for θ an MGU of H and A' , it derives $\theta(\beta) \wedge \theta(\beta') \rightarrow \theta(H')$.

The intuition is that the rule applies in two situations. The first is where H' is Skolem-free and the atom A' in τ' that gets resolved with the head atom of τ is a guard for the body. Such a rule may introduce Skolems originating in H in the body of the resolvent. The second case is where A' contains such a Skolem symbol. In this case, the resolvent will have fewer Skolems in its body than τ' . Successive applications of the second case can produce rules with no Skolem in the body, and eventually can produce a Datalog rule.

Though the Skolem rewriting can improve exponentially on ExbDR, there are also examples that give an exponential gap in the other direction. Examples can be found in the full paper [24].

Combining Several SkDR Steps Into One. The SkDR algorithm can produce many rules with Skolem symbols in the body, which is the main reason why it can lose to ExbDR. We next present the HypDR algorithm. It uses the *hyperresolution* inference rule [25] as a kind of “macro” to combine several SkDR steps into one. Our experiments show that this can be beneficial.

Definition 5.6. *The Hyperresolution Rewriting inference rule HypDR takes Skolemized single-headed rules*

$$\tau_1 = \beta_1 \rightarrow H_1 \quad \dots \quad \tau_n = \beta_n \rightarrow H_n, \quad \text{and} \quad \tau' = A'_1 \wedge \dots \wedge A'_n \wedge \beta' \rightarrow H'$$

with $n \geq 1$ such that

- for each i with $1 \leq i \leq n$, conjunction β_i is Skolem-free and atom H_i contains a Skolem symbol,
- rule τ' is Skolem-free,

and, for θ an MGU of H_1, \dots, H_n and A'_1, \dots, A'_n , if conjunction $\theta(\beta')$ is Skolem-free, it derives $\theta(\beta_1) \wedge \dots \wedge \theta(\beta_n) \wedge \theta(\beta') \rightarrow \theta(H')$.

We emphasize that all presented algorithms can be proven correct using the completeness criterion of Proposition 4.3, relying on the one-pass chase. Details are in [24]. There, we also prove the complexities of the algorithms:

Theorem 5.7. *Each presented rewriting can be computed in $2EXPTIME$, in $EXPTIME$ for bounded a , where a is the maximum relation arity in Σ , and in $PTime$ if Σ is fixed (data complexity).*

The resulting rewritings can thus be large in the worst case. From a theoretical point of view, checking fact entailment via our approach is worst-case optimal [1].

In the full paper [24], we show empirically that rewritings are suitable for practical use by using a comprehensive collection of 428 synthetic and realistic inputs. We show that our algorithms can indeed rewrite complex GTGDs, and that the rewriting can be successfully processed by modern Datalog systems. Complete evaluation results are available online [14].

6. Conclusion

We overviewed an approach for fact entailment for guarded TGDs by rewriting the GTGDs into a Datalog program that entails the same base facts on each instance. We base this on a connection between Datalog rewriting and an analysis of a specialized version of the chase. This connection allowed us to arrive at our algorithms as well as to prove their correctness. We believe this connection also makes it more intuitive why Datalog-rewritability holds. We briefly presented several algorithms based on this approach. In the full paper [24], we provide not only full proofs but also discuss implementation and experimental evaluation of the approaches. The evaluation shows that the algorithms other than FullDR are competitive and useful in practice.

In the future, we plan to generalize our framework to wider classes of TGDs, such as frontier-guarded TGDs, and to provide rewritings for conjunctive queries under certain answer semantics. Moreover, we shall investigate whether the extension of our framework to disjunctive GTGDs [19] can be used to obtain practical algorithms to rewrite into disjunctive Datalog.

References

- [1] T. Lukasiewicz, A. Cali, G. Gottlob, A general datalog-based framework for tractable query answering over ontologies, *Journal of Web Semantics* 14 (2012) 57–83.
- [2] M. O. Rabin, Decidability of second-order theories and automata on infinite trees, *Transactions of the American Mathematical Society* 141 (1969) 1–35.
- [3] V. Bárány, B. ten Cate, L. Segoufin, Guarded negation, *J. ACM* 62 (2015) 1–26.
- [4] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), *The Description Logic Handbook: Theory, Implementation and Applications*, 2nd ed., Cambridge University Press, 2007.

- [5] C. Hirsch, Guarded logics: Algorithms and bisimulation, 2002. Available at <http://www.umbriologic.com/hirsch-thesis.pdf>.
- [6] B. Marnette, Resolution and datalog rewriting under value invention and equality constraints, arXiv preprint arXiv:1212.0254 (2012). <http://arxiv.org/abs/1212.0254>.
- [7] J.-F. Baget, M.-L. Mugnier, S. Rudolph, M. Thomazo, Walking the complexity lines for generalized guarded existential rules, in: IJCAI, 2011.
- [8] G. Gottlob, S. Rudolph, M. Simkus, Expressiveness of guarded existential rule languages, in: PODS, 2014.
- [9] V. Bárány, M. Benedikt, B. Ten Cate, Rewriting Guarded Negation Queries, in: MFCS, 2013.
- [10] B. Motik, Reasoning in description logics using resolution and deductive databases, Ph.D. thesis, Karlsruhe Institute of Technology, Germany, 2006. URL: <http://digi bib.ubka.uni-karlsruhe.de/volltexte/1000003797>.
- [11] S. Ahmetaj, M. Ortiz, M. Simkus, Rewriting guarded existential rules into small datalog programs, in: ICDT, 2018.
- [12] U. Hustadt, B. Motik, U. Sattler, Reasoning in description logics by a reduction to disjunctive datalog, JAR 39 (2007) 351–384.
- [13] U. Hustadt, B. Motik, U. Sattler, Reducing SHIQ-description logic to disjunctive datalog programs., KR (2004).
- [14] M. Benedikt, M. Buron, S. Germano, K. Kappelmann, B. Motik, Guarded saturation, 2021. URL: <https://krr-oxford.github.io/Guarded-saturation/>.
- [15] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, R. Rosati, Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family, JAR 39 (2007) 385–429.
- [16] A. Cali, D. Lembo, R. Rosati, Query rewriting and answering under constraints in data integration systems, in: IJCAI, 2003.
- [17] Z. Wang, P. Xiao, K. Wang, Z. Zhuang, H. Wan, Query answering for existential rules via efficient datalog rewriting, in: IJCAI, 2021.
- [18] A. Amarilli, M. Benedikt, When can we answer queries using result-bounded data interfaces?, in: PODS, 2018.
- [19] K. Kappelmann, Decision Procedures for Guarded Logics, CoRR abs/1911.03679 (2019).
- [20] M. Y. Vardi, Why is modal logic so robustly decidable?, in: DIMACS Series in Disc. Math. and TCS, volume 31, 1997, pp. 149–184.
- [21] H. Andréka, J. van Benthem, I. Németi, Modal languages and bounded fragments of predicate logic, J. Phil. Logic 27 (1998) 217–274.
- [22] D. S. Johnson, A. C. Klug, Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies, JCSS 28 (1984) 167–189.
- [23] J. A. Robinson, A machine-oriented logic based on the resolution principle, JACM 12 (1965) 23–41.
- [24] M. Benedikt, M. Buron, S. Germano, K. Kappelmann, B. Motik, Rewriting the infinite chase, in: VLDB, 2022.
- [25] L. Georgieva, U. Hustadt, R. A. Schmidt, Hyperresolution for guarded formulae, Journal of Symbolic Computation 36 (2003) 163–192. First Order Theorem Proving.