Efficient Datalog Rewriting for Query Answering in TGD Ontologies

Zhe Wang, Peng Xiao, Kewen Wang, Zhiqiang Zhuang, Hai Wan

Abstract—Tuple-generating dependencies (TGDs) are an expressive constraint language for ontology-mediated query answering and thus query answering is of high complexity. Existing systems based on first-order rewriting methods can lead to queries too large for DBMS to handle. It is shown that Datalog rewriting can result in more compact queries, yet previously proposed Datalog rewriting methods are mostly inefficient for implementation. In this paper, we fill the gap by proposing an efficient Datalog rewriting approach for answering conjunctive queries over TGDs, and identify and combine existing fragments of TGDs for which our rewriting method terminates. We implemented a prototype system Drewer, and experiments show that it is able to handle a wide range of benchmarks in the literature. Moreover, Drewer shows superior performance over state-of-the-art systems on both the compactness of rewriting and the efficiency of query answering.

Index Terms—Query rewriting, ontology, tuple-generating dependency, existential rules.

1 INTRODUCTION

In the ontology-based data access framework [1], ontologies are used as virtual schemas over data to enrich query answering with ontological knowledge through logical reasoning, while making use of practical database management systems to scale over very large data. Tuple generating dependencies (TGDs, a.k.a. existential rules and Datalog+) as an expressive database constraint languages have recently been proposed as an ontology language for accessing data [2], [3], [4]. It has attracted much interest lately due to its expressive power covering Datalog and many Horn description logics, including the core dialects of DL-Lite [1] and $\mathcal{EL}$ [5], which underlay the OWL 2 Profiles [6]. This makes TGDs an appealing formalism for ontology-mediated query answering [7]. While query answering is undecidable over the full formalism, several interesting fragments that support tractable query answering have been proposed [2], [3], [8], [9].

There are two major approaches for query answering over ontologies expressed in TGDs or Horn description logics, the chase-based approach and the query rewriting approach. The chase-based approach relies on the termination of the chase procedure (a.k.a. forward chaining), and it is shown that computing a full chase can be rather inefficient when query answers depend only on a small portion of it [10]. On the other hand, the query rewriting approach does not require expanding the data. Given an ontology $\Sigma$ and a query $q$, a rewriting method transforms them into another query $q_{\Sigma}$, which is sometimes in a different query formalism, such that answering $q_{\Sigma}$ can be handled by conventional database management systems (DBMS) and at the same time preserves the answers to the original ontology-mediated query. The rewriting approach is particularly promising as it allows ontology-mediated query answering to be implemented on top of existing highly-optimised database query engines. While many algorithms and systems have been developed for various description logics [11], [12], [13], [14], particularly for DL-Lite and $\mathcal{EL}$ [15], [16], [17], [18], [19], it is challenging to extend them to more general TGDs, which allow predicates of arbitrary arities (instead of only unary and binary predicates) and variable permutations in the rules.

Existing query rewriting systems for TGDs are typically based on first-order rewritings [20], [21], [22], i.e., the query $q_{\Sigma}$ is formulated as a first-order formula over a DBMS. Efficient rewriting systems have been developed [20], [21], with several optimisations implemented to reduce the sizes of the results of rewriting. A limitation of such an approach is that it can only handle ontologies and queries that are first-order rewritable. Well-accepted first-order rewritable classes are the linear and sticky TGDs [3]. Yet many practical ontologies do not necessarily fall into these classes, such as some ontologies formulated in $\mathcal{EL}$. Even for ontologies and queries that are first-order rewritable, the results of rewriting can suffer from a significant blow up and become difficult for DBMS to handle [19], [23].

On the other hand, taking Datalog as the target query language can lead to much more compact rewritings, i.e., the resulting Datalog programs have relatively small sizes, and it is shown for description logics that executing Datalog rewritings is much more feasible for DBMS than equivalent first-order rewritings [18]. All ontologies and queries that are first-order rewritable are trivially Datalog rewritable, and more Datalog rewritable classes are known, such as the guarded TGDs [24]. However, existing research on Datalog rewriting of TGDs are mostly theoretical. For instance, it is shown that conjunctive queries with guarded TGDs can be rewritten into Datalog queries [24], [25], whereas those with guarded disjunctive TGDs can be rewritten into disjunctive Datalog queries [7]; and a method of polynomial-sized rewritings for guarded (disjunctive) TGDs and atom queries is proposed in [26]. Rewriting methods into polynomial-
sized non-recursive Datalog queries have been proposed for linear (and sticky) TGDs [27], [28], [29]. The construction methods in the above works are primarily of theoretical interest, and no practical system has been implemented based on them. While several algorithms and systems have been developed for Datalog rewriting for various description logics [11], [12], [17], [18], [19], [23], very few systems have been developed for Datalog rewriting over TGDs that are more general than description logics. A notable exception is ChaseGoal [10], which however, relies on the termination of the chase procedure.

In this paper, we fill the gap by presenting both a practical approach and a prototype system for Datalog rewriting and query answering over a wide range of ontologies expressed in TGDs. Our algorithm is based on the notion of unfolding [30] and to achieve compactness of rewriting, we separate the results of unfolding into short rules by introducing the so-called separating predicates and reusing such predicates when possible. While such a rewriting process may not terminate, we move on to identify classes of ontologies where the rewriting process terminates, introducing a class by combining existing well-accepted classes. And we introduce an efficient algorithm for computing the Datalog rewritings. Finally, we implemented a prototype system, Drewer, and experiments show that it is able to handle a wide range of benchmarks in the literature. Moreover, Drewer shows superior performance over state-of-the-art systems on both the compactness of rewriting and the efficiency of query answering.

This paper is a significant extension of a conference paper presented at IJCAI-2020 [31]. As the new class BExp of TGD ontologies is introduced, major theoretical results are extended and revised. Also, all proofs of these results are provided. Moreover, experimental results have been significantly enhanced. In particular, more benchmarks are included, and results in Table 1 and Figures 1–3 are new.

2 Preliminaries

Let \( C, N, \) and \( V \) be pairwise disjoint, countably infinite sets of respectively, constants, labelled nulls (i.e., place-holders for unknown values and thus can be seen as existentially quantified variables), and variables. We use \( X \) to denote a sequence (or a set, with a slight abuse of notation) of variables \( X_1, \ldots, X_n \in V \). A term is either a constant, a null, or a variable.

A relational schema \( \mathcal{R} \) defines a set of relational symbols (or predicates), where each predicate \( A \) is associated with an arity \( n \), denoted \( A/n \). An atom \( A \) is of the form \( A(t_1, \ldots, t_n) \) where \( t_1, \ldots, t_n \) are terms. For an atom \( A, C, \alpha, N, \) and \( V_\alpha \) denote respectively, the sets of constants, nulls, and variables in \( \alpha \), which can be naturally extended to a set of atoms. A fact \( t \) is an atom where \( t \in C \) for all \( 1 \leq i \leq n \). An instance \( I \) for a schema \( \mathcal{R} \) is a (possibly infinite) set of atoms, and a dataset is a finite set of facts. For convenience, a finite set of atoms can often be identified with their conjunction.

A substitution \( \pi \) is a function from a set \( T \) of variables and nulls to a set \( T' \) of terms. By \( t \pi = t' \), we denote that \( t \in T \) is mapped to \( t' \in T' \) by \( \pi \). If \( T = \{ t_1, \ldots, t_n \} \) and \( t_i \pi = t'_i \) for \( i = 1, \ldots, n \), then the substitution \( \pi \) is also denoted as the (possibly empty) set \( \{ t_1 \mapsto t'_1, \ldots, t_n \mapsto t'_n \} \). Sometimes, by defining \( t \pi = t \) for \( t \notin T \), \( \pi \) is also regarded as a mapping on the set of all terms. For an atom \( A(t_1, \ldots, t_n) \), \( A(t_1 \pi, \ldots, t_n \pi) \) denotes the atom \( A(t_1, \ldots, t_n) \) and it naturally extends to sets of atoms and formulas. For a subset \( t \) of \( T \), \( \pi|_t \) denotes the substitution \( \{ t \mapsto t' \in \pi \land t \in T \} \). For two substitutions \( \pi_1 \) and \( \pi_2 \), their composition is the substitution \( \pi_1 \pi_2 \) such that \( t(\pi_1 \pi_2) = (t \pi_1) \pi_2 \) for each \( t \) in the domain of \( \pi_1 \). Given two sets \( A \) and \( A' \) of atoms, a homomorphism \( \pi \) from \( A \) to \( A' \) is a substitution from terms in \( A \) to terms in \( A' \) such that \( A \pi \subseteq A' \). A unifier between two sets of atoms \( A \) and \( A' \) is a substitution \( \tau \) such that \( A \tau = A' \); and \( \tau \) is a most general unifier (MGU) if for each unifier \( \tau' \) between \( A \) and \( A' \), there exists a substitution \( \pi \) such that \( \tau' = \pi \tau \).

A tuple-generating dependency (TGD) \( \sigma \) over a schema \( \mathcal{R} \), also referred to as an existential rule (or simply, a rule), is a first-order formula of the form

\[
\forall X. \forall Y. [\exists Z. \varphi(X, Z) \leftarrow \psi(X, Y)]
\]

where \( X, Y, Z \) are pairwise disjoint vectors of variables \( X \cup Y \cup Z \subset V \), and \( \varphi(X, Z) \) and \( \psi(X, Y) \) are conjunctions of atoms over \( \mathcal{R} \) containing only variables from respectively \( X \cup Z \) and \( X \cup Y \). Variables in \( X \) are frontier variables and those in \( Z \) are existential variables, and we use \( X_\alpha \) and \( Z_\sigma \) to emphasise that they are for TGD \( \sigma \). Formula \( \varphi \) is the head of \( \sigma \), denoted \( \text{head}(\sigma) \), and formula \( \psi \) is the body of \( \sigma \), denoted \( \text{body}(\sigma) \) (again, they can be seen as sets of atoms). For brevity, universal quantifiers in a TGD are often omitted, and we assume each TGD employs a distinct set of variables.

A Datalog rule is a TGD with a single head atom and no existential variable. An ontology is a finite set of TGDs.

An instance \( I \) satisfies a TGD \( \sigma \), denoted \( I \models \sigma \), if whenever a homomorphism \( \pi \) exists from \( \text{body}(\sigma) \) to \( I \), there exists a homomorphism \( \pi' \) from \( \text{head}(\sigma) \) to \( I \) with \( \pi|_{X_\alpha} \subseteq \pi' \). An instance \( I \) satisfies a set of TGDs \( \Sigma \), denoted \( I \models \Sigma \), if \( I \models \sigma \) for each \( \sigma \in \Sigma \). A TGD \( \sigma \) (or a set of TGDs \( \Sigma \)) implies another TGD \( \sigma' \) if for each instance \( I \models \sigma \) (resp., \( I \models \Sigma \)), \( I \models \sigma' \).

A conjunctive query (CQ) over \( \mathcal{R} \) is a first order formula of the form \( \exists Y. \psi(X, Y) \), where \( \psi(X, Y) \) is a conjunct of atoms over \( \mathcal{R} \) whose variables are in \( X \cup Y \). Such a CQ will be conveniently represented as a Datalog rule \( q : Q(X) \leftarrow \psi(X, Y) \) in the rest of this paper, where the query predicate \( Q \) for \( q \) is a predicate with arity \( |X| \) but not in \( \mathcal{R} \). A union of conjunctive query (UCQ) is a finite set of CQs, which can be seen as a finite set of Datalog rules \( Q \) with the same predicate \( Q \) in the head. Throughout the paper, we will use \( Q \) to denote the query predicate for a query, which should not be confused with a UCQ \( Q \) that is a set of Datalog rules.

The answers of a CQ \( q \) on a dataset \( D \) are defined as the set \( q(D) = \{ X \pi \mid \pi \text{ is a homomorphism from } \text{body}(q) \to D \} \). Thus, the answers of a UCQ \( Q \) over \( D \) are the set \( Q(D) = \bigcup_{q \in Q} q(D) \).

An ontology-mediated query (OMQ) \( \Sigma Q \) consists of a finite set of TGDs \( \Sigma \) over \( \mathcal{R} \) and a UCQ \( Q \) (a set of Datalog rules), which can be seen as a set of rules \( \Sigma \cup Q \). A Datalog query is an OMQ where \( \Sigma \) is a set of Datalog rules, which again, can be conveniently seen as a set of Datalog rules. For a dataset \( D \) over \( \mathcal{R} \), the answers of an OMQ are those so-called certain answers. We formally define this notion as follows.

An instance \( I \) is a model of \( D \) w.r.t. \( \Sigma \), denoted \( I \models \Sigma \cup D \), if \( D \subseteq I \) and \( I \models \Sigma \). It is shown in [32], a universal model (a.k.a.
canonical model) \( I \) of \( D \) w.r.t. \( \Sigma \) always exists such that for each model \( I' \) of \( D \) w.r.t. \( \Sigma \), there is a homomorphism \( \pi \) with \( I \pi \subseteq I' \). Clearly, the universal model is unique up to homomorphism. The set of answers of an OMQ \( \Sigma_q \) on \( D \) is defined as \( \Sigma_q(D) = Q(I) \) where \( I \) is the universal model of \( D \) w.r.t. \( \Sigma_q \).

Two OMQs \( \Sigma_q \) and \( \Sigma_{q'} \) are equivalent if \( \Sigma_q(D) = \Sigma_{q'}(D) \) for each dataset \( D \) over \( R \). A Datalog rewriting of an OMQ \( \Sigma_q \) is a Datalog query \( \Pi_q' \), where \( Q \) and \( Q' \) have the same query predicate \( Q \), such that \( \Sigma_q \) and \( \Pi_q' \) are equivalent, that is, \( \Sigma_q(D) = \Pi_q'(D) \) for each dataset \( D \) over \( R \). When \( \Pi_q' \) is a UCQ, it is also referred to as a UCQ rewriting of \( \Sigma_q \). The query answering of OMQs can be reduced to that of answering \( \Sigma_q \) with \( Q \) consisting of a single CQ \( q \), and hence in what follows, w.l.o.g., we consider only OMQs with single CQs, denoted \( \Sigma_q \).

In the next section, we introduce a compact Datalog rewriting approach.

### 3 Compact Datalog Rewriting

Existing approaches on UCQ rewriting of OMQs are (essentially) based on the notion of piece unification [2], [33]. First, for an instance \( I \) and its subset \( I' \), a variable occurring in an atom in \( I' \) but not in any atoms in \( I \), \( I' \) is an exclusive variable for \( I \) in \( I \). For a Datalog rule \( \rho \) and a TGD \( \sigma \), a piece unifier of \( \rho \) and \( \sigma \) is a tuple \( \mu = (B, \tau, \rho) \), where \( \emptyset \subseteq B \subseteq \text{body}(\rho) \), \( H \subseteq \text{head}(\sigma) \), and \( \tau \) is a minimal MGU between \( B \) and \( H \) such that for each existential variable \( Z \in Z_\sigma \) and each term \( t \) other than \( Z \) that is unified with \( Z \), \( \tau \) must be an exclusive variable for \( B \) in \( \text{body}(\rho) \). Intuitively, it excludes the cases where \( Z \) is unified with a constant, with a variable in \( \text{head}(\sigma) \) other than \( Z \), or with a variable in \( \text{body}(\rho) \) shared between atoms inside and outside \( B \). Also, the minimality condition ensures variables outside \( B \) are not unnecessarily unified. In this case, the result of unfolding \( \rho \) by \( \sigma \) with \( \mu \) [30], [33], denoted \( \sigma^{\rho}(\mu) \), is

\[
\text{head}(\rho) \tau \leftarrow \bigwedge (\text{body}(\rho) \setminus B) \tau \land \bigwedge \text{body}(\sigma) \tau. \tag{+}
\]

**Example 1.** Consider a TGD \( \sigma \):

\[\exists Z.A(X, Y, Z) \leftarrow B(X, Y)\]

and a CQ \( q \):

\[Q \leftarrow A(U, U, V) \land A(U, V, V) \land A(U, U, W).\]

Then, \( \mu_1 = \langle \{A(U, V, W)\}, \{A(X, Y, Z)\}, \{X \rightarrow U, Y \rightarrow V, Z \rightarrow W\} \rangle \) is not a piece unifier, as existential variable \( Z \) is unified with \( W \) that is shared with \( A(U, U, W) \). Neither is \( \mu_2 = \langle \{A(U, V, W)\}, \{A(X, Y, Z)\}, \{X \rightarrow U, Y \rightarrow V, Z \rightarrow W\} \rangle \) a piece unifier, as it forces \( Z \) to be unified with another variable \( Y \) in the same rule head. Yet, \( \mu_3 = \langle \{A(U, V, W), A(U, U, W)\}, \{A(X, Y, Z)\}, \{V \rightarrow U, X \rightarrow U, Y \rightarrow U, Z \rightarrow W\} \rangle \) is a piece unifier, and \( \sigma(q, \mu_3) = Q \leftarrow A(U, U, U) \land B(U, U) \).

For a UCQ \( Q \), a cover of \( Q \) is a minimal subset \( Q' \subseteq Q \) such that for each CQ \( q \in Q \), there is a CQ \( q' \in Q' \) that implies \( q \). Note that the cover of a UCQ \( Q \) is unique up to query equivalence, and we denote it as \( \text{cover}(Q) \). An unfolding sequence of an OMQ \( \Sigma_q \) is a sequence of UCQs \( Q^i (i \geq 0, \text{with the same query predicate}) \), where \( Q^0 = \{q\} \) and \( Q^{i+1} = \text{cover}(Q^i \cup \{\sigma(q', \mu) | \sigma \in \Sigma, q' \in Q^i, \mu \text{ is a piece unifier of } q' \text{ and } \sigma\}) \). It is shown that a UCQ rewriting of \( \Sigma_q \) exists if there is some \( k \geq 0 \) such that \( Q^{k+1} \) is equivalent to \( Q^k \); and in this case, \( Q^k \) is a UCQ rewriting of \( \Sigma_q \) [20], [22].

As we have discussed before, for some OMQ, the size of its UCQ rewriting can be very large, containing a huge number of CQs and some CQs can be very long, which makes it difficult to be handled by DBMS [19], [23]. In what follows, we introduce an approach for compact Datalog rewriting by splitting long Datalog rules generated via rewriting into shorter ones. As a first step, we present an alternative rewriting operator of a Datalog rule by a TGD.

**Definition 1.** For a Datalog rule \( \rho \) and a TGD \( \sigma \) and a piece unifier \( \mu = (H, B, \tau) \) of \( \rho \) and \( \sigma \), the result of rewriting \( \rho \) by \( \sigma \), denoted \( \sigma^* \), consists of \( \sigma(\rho, \mu) \) and the following two Datalog rules:

\[
\begin{align*}
\text{head}(\rho) \tau & \leftarrow \bigwedge (\text{body}(\rho) \setminus B) \tau \land P(X), \tag{1} \\
\text{body}(\sigma) \tau & \leftarrow \bigwedge \text{body}(\sigma) \tau, \tag{2}
\end{align*}
\]

where \( X = (X_\rho \cup \forall \text{body}(\rho) \setminus B) \tau \cap X_\sigma \tau \cdot P \) is a fresh predicate with arity \( |X| \), called a separating predicate.

To avoid ambiguity on the sequence of variables in \( X \), we assume a pre-order on the variables in \( V \), and the order of variables in \( X \) follows such a pre-order. Intuitively, we split the body of \( \sigma(\rho, \mu) \) by introducing \( P \) to obtain compact rules. Note that rule \( \sigma(\rho, \mu) \) can be obtained by unfolding (1) by (2), yet it is generated for the correctness of rewriting as we show later. We call rules of the form \( \sigma(\rho, \mu) \) auxiliary rules; they will be deleted after the whole rewriting process is completed.

**Example 2.** For the CQ \( q \) and the TGD \( \sigma \) in Example 1, \( \sigma^* \) consists of the following Datalog rules:

\[
\begin{align*}
Q & \leftarrow A(U, U, U) \land B(U, U), \\
Q & \leftarrow A(U, U, U) \land P(U), \\
P(U) & \leftarrow B(U, U).
\end{align*}
\]

Note that \( X_\rho = \emptyset, \forall \text{body}(\rho) \setminus B \tau = \{U\} \), and \( X_\sigma \tau = \{U\} \). Hence, the separating predicate is a unary one.

For an OMQ, its rewriting sequences can be defined using \( \sigma^* \). To avoid repetitively introducing fresh separating predicates, we note that it is possible to reuse them. This is achieved through a labelling function \( \lambda(\cdot) \), such that for the separating predicate \( P \) in Definition 1, \( \lambda(P) = (B \tau, X) \). Intuitively, the label records how \( P \) is introduced (i.e., the atoms that are unified in the rewriting). If a rewriting step introduces a new separating predicate \( P' \) with the same arity as \( P \) such that \( \lambda(P') \) is equivalent to \( \lambda(P) \) up to variable renaming, then \( P \) is reused to replace \( P' \).

The notion of \( \text{cover}(Q) \) can be extended to Datalog queries \( II \). Note that one can define a cover of \( II \) to be a minimal subset that implies all the Datalog rules in \( II \), yet such a definition would eliminate auxiliary rules (as they are implied by the separated rules). Hence, we will directly extend the original definition of \( \text{cover}(Q) \) to \( \text{cover}(II) \) which eliminates a Datalog rule only if it is implied by another rule. We are ready to define our Datalog rewriting.
Definition 2. A rewriting sequence of an OMQ $\Sigma_q$ is a sequence of Datalog queries $\Pi^i$ ($i \geq 0$, with the same query predicate), where $\Pi^0 = \{q\}$ and
$$\Pi^{i+1} = \text{cover}(\Pi^i \cup \bigcup_{\sigma \in \sigma^*(\rho, \mu), \mu \text{ is a piece unifier of } \rho \text{ and } \sigma} \Sigma_{\rho \in \Pi^1}),$$
for $i > 0$, with separating predicates being reused whenever possible.

For the convenience of discussion, we assume fixed orders on the rules involved in the rewriting process and on the introduction of separating predicates, which allows us to consider a unique rewriting sequence. The rewriting of $\Sigma_q$, denoted as $\text{rew}(\Sigma_q)$, is obtained from $\Pi^k$ for the smallest $k \geq 0$ such that $\Pi^{k+1}$ is equivalent to $\Pi^i$ or from $\Pi^\infty$ if such a $k$ does not exist, by deleting all the auxiliary rules.

Example 3. Consider the CQ $q$ in Example 1 and another TGD $\sigma_1$:
$$\exists Y_1. A(X_1, X_1, Y_1) \leftarrow B(X_1, Z_1).$$
Then, there is a piece unifier $\mu_4$ of $q$ and $\sigma_1$ with $\mu_4 = \langle \{A(U, V, W), A(U, U, W), \{A(X_1, X_1, Y_1), \{V \rightarrow U, X_1 \rightarrow U, Y_1 \rightarrow W\}\}\rangle$, and $\sigma_1^*(q, \mu)$ consists of:
$$\rho_1: q \leftarrow A(U, U, U) \land B(U, Z_1), \quad \rho'_1: q \leftarrow A(U, U, U) \land P(U), \quad \rho''_1: P(U) \leftarrow B(U, Z_1).$$

The rewriting path $P$ is the same as in Examples 1 and 2 and has been reused, with $A(P) = \langle \{A(U, U, W), \{U\}\rangle$.

Furthermore, consider another TGD $\sigma_2$:
$$\exists X_2. [A(X_2, X_2, X_2) \land B(X_2, Y_2)] \leftarrow C(Y_2).$$
A piece unifier of $\rho_1$ and $\sigma_2$ is $\mu_5 = \langle \{A(U, U, U), B(U, Z_1), \{A(X_2, X_2, X_2), B(X_2, Y_2), \{X_2 \rightarrow U, Y_2 \rightarrow Z_1\}\}\rangle$, and $\sigma_2^*(\rho_1, \mu)$ consists of:
$$\rho_2: q \leftarrow C(Z_1), \quad \rho'_2: q \leftarrow P', \quad \rho''_2: P' \leftarrow C(Z_1).$$

Based on the observation that our rewriting will terminate if no auxiliary rules are not generated, we introduce a new class Sep of TGD ontologies and show that Sep is a proper superset of Shy. To cover more useful Datalog rewritable cases, we introduce another class BFin of TGD ontologies, which properly contains Lin, Shy, and BFin as its subclasses. Moreover, we identify a concrete class of BFin, called block expandable and denoted BExp, which contains these four concrete classes too.

4.1 Datalog Rewriting of FUS

It is known that the unfolding of a CQ always terminates on the class of finite unification sets (FUS) of TGDs [2]. We show that our Datalog rewriting terminates for FUS too.

Proposition 2. For an OMQ $\Sigma_q$ whose ontology is FUS, $\text{rew}(\Sigma_q)$ is finite.

To prove Proposition 2, we first introduce the notion of a rewriting path which results in a Datalog rule generated during a sequence of rewriting. A rewriting path of an OMQ $\Sigma_q$ is defined inductively as follows: $q$ is a rewriting path whose result is itself; and if $\theta$ is an rewriting path whose result is $\rho$ and $\rho$ can be rewritten by a TGD $\sigma \in \Sigma$ with piece unifier $\mu$, then $\theta \mu_{\sigma^1}, \theta \mu_{\sigma^2}, \theta \mu_{\sigma^3}$ and $\theta \mu_{\sigma^4}$ are three rewriting paths resulting from the three Datalog rules in $\sigma^*(\rho, \mu)$ of the forms respectively $(+)$, $(1)$, and $(2)$ in Definition 1. For convenience, we may use a rewriting path to represent its resulting Datalog rule.

For a rewriting path $\theta$, two TGDs $\sigma_1, \sigma_2$, two piece unifiers $\mu_1, \mu_2$, and some $\epsilon_1, \epsilon_2 \in \{+, (1), (2)\}$, we have the following useful observations:

(A) If $\theta \mu_1 \sigma_1^{\epsilon_1} \mu_2 \sigma_2^{\epsilon_2}$ is a rewriting path, then so is $\theta \mu_1 \sigma_2^{*\epsilon_1} \mu_2 \sigma_2^{*\epsilon_2}$. It can be seen from the fact that the body of $\theta \mu_1 \sigma_1^{*\epsilon_1}$ contains all the non-separating body atoms in $\theta \mu_1 \sigma_1^{\epsilon_1}$.

(B) If $\theta \mu_1 \sigma_1^{(1)} \mu_2 \sigma_2^{(2)}$ is a rewriting path, then so is $\theta \mu_2 \sigma_2^{(1)}$, where $\mu_2$ is similar to $\mu_2$ with its MGU composed with that of $\mu_1$. It can be seen from the fact that the body of $\theta$ contains all the non-separating body atoms.
in \( \theta \mu_1 \sigma_1^{(1)} \). And the body of \( \theta \mu_2 \sigma_2^{(2)} \) contains all the non-separating body atoms in \( \theta \mu_1 \sigma_1^{(1)} \mu_2 \sigma_2^{(2)} \).

(C) If \( \theta \mu_1 \sigma_1^{(2)} \mu_2 \sigma_2^{(2)} \) is a rewriting path, then it has the same body as \( \sigma_1 \mu_1 \mu_2 \sigma_2 \), where \( \tau_1 \) is the MGU of \( \mu_1 \).

For a rewriting path \( \theta = q_1 \mu_1 \sigma_1^{(1)} \cdots \mu_n \sigma_n^{(n)} \) for some \( n \geq 0 \), \( \theta^* \) is obtained from \( \theta \) by replacing \( \epsilon_1, \ldots, \epsilon_i \) with \((*)\). By a simple induction, \( \theta^* \) is also a rewriting path; and indeed, it is a sequence of unfolding. Also, let \( \theta^- \) be obtained from \( \theta \) by removing \( \mu_i \sigma_i^{(i)} \) from the sequence for all \( \epsilon_i = (1) \), then \( \theta^- \) is also a rewriting path, and the body of \( \theta^- \) contains all the non-separating body atoms in \( \theta \).

Proof of Proposition 2. We first show that the size of each Datalog rule in \( \text{rew}(\Sigma_q) \) is bounded. Since \( \Sigma \) belongs to FUS, any OMQ with \( \Sigma \) as the ontology is UCQ rewritable. Let \( m \) be the maximum size of a CQ in a UCQ rewriting of \( \Sigma_q \) or of \( \Sigma \cup \{ \mathcal{Q} \leftarrow \text{body}(\sigma) \} \) for some \( \sigma \in \Sigma \). We want to show that the size of each Datalog rule in \( \text{rew}(\Sigma_q) \) is at most \( m \). Towards a contradiction, suppose a Datalog rule \( \rho \in \text{rew}(\Sigma_q) \) has size larger than \( m \). Then \( \rho \) is the result of a rewriting path of the form \( q_1 \mu_1 \sigma_1^{(1)} \cdots \mu_n \sigma_n^{(n)} \) where \( n \geq 0 \), \( \sigma_1, \ldots, \sigma_n \) are TGDs in the ontology \( \Sigma_q \), \( \mu_1, \ldots, \mu_n \) are piece unifiers, and \( \epsilon_1, \ldots, \epsilon_n \in \{ (\ast), (1), (2) \} \).

Suppose \( \epsilon_i = (2) \) for some \( 1 \leq i \leq n \) and we take the maximum \( i \) satisfying this condition; otherwise if there is not such an \( i \), then we simply take \( i = 1 \). Let \( q' = \mathcal{Q} \leftarrow \text{body}(\sigma_i) \). From Observation (C), the body of \( \rho \) is the same as that of \( \theta = q' \mu_1 \sigma_1^{(1)} \cdots \mu_i \sigma_i^{(i)} \cdots \mu_n \sigma_n^{(n)} \).

Note that \( \theta^- \) is generated during unfolding of \( \Sigma \cup \{ q' \} \), whose body contains all the non-separating body atoms of \( \rho \), and thus has a size greater than \( m \). Hence, \( \theta^- \) must be eliminated (from the UCQ rewriting of \( \Sigma \cup \{ q' \} \)) due to being implied by some other Datalog rule \( \theta' \) generated during the unfolding. For the implication to hold, the body of \( \theta' \) can be homomorphically mapped to that of \( \theta^- \). Hence, in the sequence to generate \( \rho \), if we replace the sequence after \( \sigma_i^{(i)} \) with the fragment of \( \theta' \) after \( q' \), it is still a rewriting path, and the resulting Datalog rule has the same head as \( \rho \) and has a body that can be homomorphically mapped to that of \( \rho \). Hence, \( \rho \) should be eliminated, which is a contradiction.

We have shown that the sizes of Datalog rules in \( \text{rew}(\Sigma_q) \) are bounded. Then, it suffice to note that the introduced separating predicates are also bounded due to predicate reuse and the fact that the possible labels \( \lambda(P) \) are also bounded by the head atoms of TGDs in \( \Sigma \).

4.2 The Separable Class

Yet, some common Datalog rewritable ontologies are not in FUS. For example, the ontology \( \Sigma_{tr} \) with a single TGD that defines a relation satisfying transitivity, \( A(X, Y) \leftarrow A(X, Z) \land A(Z, Y) \), is not in FUS. The rewriting on \( \Sigma_{tr} \) does not terminate, i.e., \( \Pi^0 \) is inequivalent to \( \Pi^{i+1} \) for \( i \leq 0 \). It is not hard to see that the termination issue is caused by the generation of infinitely many auxiliary rules. Towards a new class of Datalog rewritable ontologies, we consider a variant of rewriting where auxiliary rules are not generated. Formally, the result of \( \text{sep-rewriting} \) (or \( \text{sep-rewriting} \)) is defined in the same way as in Definition 1 except that the auxiliary rule of the form \( (\ast) \) is not generated. Then, we define \( \text{rew}_s(\Sigma_q) \) for an OMQ \( \Sigma_q \) in the same as in Definition 2 with sep-rewriting in places of rewriting.

Proposition 3. For an OMQ \( \Sigma_q \), \( \text{rew}_s(\Sigma_q) \) is always finite.

Proof. Suppose the maximum arity of predicates is \( l \), and each rule (TGD or CQ) in \( \Sigma_q \) contains at most \( m \) head atoms and \( n \) body atoms. We show that the sizes of the Datalog rules of the forms (1) and (2) generated during the construction of \( \text{rew}_s(\Sigma) \) are bounded. First, each Datalog rule generated has at most \( n \) body atoms. In particular, for rules of the form (1), noting that \( B \) contains at least one atom, and thus the rule body is at most the size of body(\( \rho \)), that is, rewriting does not increase the number of body atoms. Also, only a bounded number of separating predicates are generated due to predicate reuse and their arities are bounded by \( l \). More specifically, for each separating predicate \( P \), \( \lambda(P) \) always consists of at most \( m \) head atoms and the arity of \( P \) is bound by \( l \).

We consider atom queries (AQs), which are CQs with single body atoms, and define the class of \( \text{separable} \) ontologies for which sep-rewriting (rather than rewriting) is sufficient to produce Datalog rewritings on AQs.

Definition 3. An ontology \( \Sigma \) is \( \text{separable} \) if for each AQ \( q \), \( \text{rew}_s(\Sigma_q) \) is a Datalog rewriting of \( \Sigma_q \).

Intuitively, the condition requires the bodies of the Datalog rules generated during rewriting can be separated. Note that the bodies of CQs may not be separated in general, which is why we consider AQs here. The class of separable rule sets is denoted Sep.

Example 4. The ontology \( \Sigma_{tr} \) with a single TGD below is separable,

\[
\sigma_0 : A(X_0, Z_0) \leftarrow A(X_0, Z_0) \land A(Y_0, Z_0),
\]

and it is not hard to verify for any AQ \( q \), \( \text{rew}_s(\Sigma_{tr} \cup \{ q \}) \) is a Datalog rewriting of \( \Sigma_{tr} \cup \{ q \} \).

On the other hand, an ontology \( \Sigma_{be} \) (the meaning of the subscript will be made clear later) with the following TGDs is not separable:

\[
\sigma_1 : \exists Y_1. A(X_1, Y_1) \leftarrow B(X_1),
\]

\[
\sigma_2 : B(X_2) \leftarrow A(Y_2, X_2),
\]

\[
\sigma_3 : C(X_3) \leftarrow A(X_3, Y_3) \land A(Y_3, Z_3).
\]

To see it is not separable, consider an AQ \( q : \mathcal{Q} \leftarrow C(X), \) and a CQ generated via the unfolding of \( \Sigma_{be} \cup \{ q \} \) (by applying \( \sigma_3 \), \( \sigma_1 \), \( \sigma_2 \), and then \( \sigma_3 \)) is \( \mathcal{Q} \leftarrow B(Y) \). Yet, it cannot be captured by \( \text{rew}_s(\Sigma_{be} \cup \{ q \}) \) with the body atoms of \( \sigma_3 \) rewritten separately.

From Proposition 3, for an atom query and a separable ontology, a Datalog rewriting always exists. Yet the definition does not suggest how to effectively identify such an ontology. For a concrete class of separable ontologies, we show that the existing Shy class [9] is a subclass of Sep. We present the definition of the Shy class below.

A position is of the form \( A[i] \) with \( A \) being an \( n \)-ary predicate and \( 1 \leq i \leq n \), and a variable \( X \) occurs at position \( A[i] \) if there is an atom \( A(t_1, \ldots, t_n) \) with \( t_i = X \). For an ontology \( \Sigma \) and an existential variable \( Z \) in \( \Sigma \), a position \( A[i] \) is \( \text{invoked} \) by \( Z \) if there is a TGD \( \sigma \in \Sigma \) with
head(σ) = A(t₁, ..., tₙ) and either tᵢ = Z or tᵢ is a frontier variable that occurs in body(σ) only at positions that are invaded by Z. Recall that we assume each rule has a distinct set of variables. Then, a variable X in Σ is attacked by Z if X only occurs in positions invaded by Z. Two atoms in the same rule body are chained if (1) they share a variable that is attacked, or (2) they each contains a frontier variable and these two variables are both attacked by the same variable. Finally, Σ is shy if it does not contain two chained atoms, and we denote the class of shy ontologies as Shy.

The ontology Σ₀ₑ in Example 4 is not shy. We can see that Y₁ invades the positions A[2] in the head of σ₁, B[1] because of the frontier variable X₂, and A[1] due to the frontier variable X₁. Hence, the two body atoms in σ₃ are chained because Y₃ is attacked by Y₁.

The following result shows that every shy ontology is also separable. The class Shy was originally introduced to guarantee the termination of a special form of chase, and to the best of our knowledge, Theorem 1 is a first result on the datalog rewritability of Shy for atom queries.

**Theorem 1.** Shy ⊆ Sep.

To prove Theorem 1, we need the following lemma, which states that for an atom query and a shy ontology, in each unfolding step, the body atoms can be separately resolved.

**Lemma 2.** For a shy ontology Σ, an AQ q and the OMQ σₜ, for each CQ q’ generated during the unfolding of σₜ, if q’ can be rewritten by a TGD σ ∈ Σ with piece unifier (B, H, τ), then for each atom α ∈ B, q’ can be rewritten by σ with piece unifier of the form ⟨{α}, H’, τ’⟩ with H’ ⊆ H and τ = τ’π for some substitution π.

**Proof of Theorem 1.** For each AQ q, consider the OMQ σₜ and let Π = rewₜ(Σₜ). We want to show that for each dataset D, Π(D) = Σₜ(D). As rewₜ(Σₜ) ⊆ rew(Σₜ), Π(D) ⊆ Σₜ(D).

To show the other direction, we want to show for each unfolding path of σₜ, θ = q₁σ₁⁽¹⁾ · · · qₖσₖ⁽¹⁾, its resulting CQ can be obtained by unfolding Datalog rules (which may resolve separating predicates in Π), which would imply that Σₜ(D) ⊆ Π(D). We show this claim by an induction on k ≥ 0. Note that the Datalog rules in Π correspond to rewriting paths with superscripts ε ∈ \{(1), (2)\}.

The case of k = 0, where θ = q, trivially holds. For k = 1, θ = q₁σ₁⁽¹⁾, which can be obtained by unfolding q₁σ₁⁽¹⁾ by q₁σ₁⁽¹⁾. Assume the statement holds for k ≥ 1, we want to show it holds for k + 1. By the induction assumption, θ can be obtained by unfolding Datalog rules (represented by their rewriting paths) θ₁, ..., θₙ ∈ Π with 1 ≤ i ≤ n. That is, θ has the same result as some rewriting path θ₁θ₂ · · · θₙ⁻¹θₙ. Suppose θ can be rewritten by TGD σ ∈ Σ with piece unifier μ = (B, H, τ) with B = {α₁, ..., αₘ}. As θ corresponds to θ₁θ₂ · · · θₙ⁻¹θₙ, each αᵢ (1 ≤ j ≤ m) must occur in some body(θᵢ)τᵢ’ · · · τᵢ with 1 ≤ i ≤ n (where each τᵢ is the substitution in the piece unifier μᵢ). From Lemma 2, each θᵢ (1 ≤ j ≤ m) can be rewritten by σ with a piece unifier μᵢ’ = ⟨{αᵢ}, H, τᵢ’⟩.

We want to show that θμσ⁽¹⁾ can be obtained by unfolding Datalog rules in Π. Note that θμσ⁽¹⁾ is of the form head(θ)τ ← \(\bigwedge \{\text{body}(\theta) \setminus B\} \land \text{body}(\sigma)\)τ and it corresponds to the rewriting path \(θ₁μ₁’ \cdots μₙ₋₁θₙσ\). For each 1 ≤ j ≤ m, there are two rewriting paths \(θ₁μ₁’ \cdots μ_jσ⁽¹⁾\), which leads to a rule of the form \(ρ⁽¹⁾ = P_J(X_j) ← \text{body}(\sigma)\tau, \land θ₁μₗσ⁽²⁾\), which leads to \(ρ⁽²⁾ = head(θjτ) ← \(\text{body}(θj) \setminus \{α_j\}\)τᵢ’ · · · τᵢτ \∧ P_J(X_j)\). Clearly, \(σ⁽¹⁾\) can be unfolded by \(μ_j = ((P_J(X_j)), \{P_J(X_j), \emptyset\})\) for all 1 ≤ j ≤ n. Also, as \(θ₁j ∈ Π, θ₁μ₁’σ⁽¹⁾\) and \(θ₁μₗσ⁽²⁾\) are both in Π. Moreover, \(θₙσ⁽¹⁾\) can be obtained from the following rewriting path

\[
θ₁μ₁’τ₁’ · · · τₙ \land θ₂μ₂’τₙ \land τₙ \land \ldots \land θ₁₁τ₁₁’ · · · τₙ \land θ₂μ₂’τₙ \land τₙ \land \ldots
\]

We have shown that \(θμσ⁽¹⁾\) can be obtained by unfolding Datalog rules in Π, and that is, the statement holds for k + 1.

The strictness of the set containment can be seen from Example 5.

**Example 5.** An example of a separable but not shy ontology Σₜₑₙₚₚ consists of the following TGDs:

\[
σ₁ : ∀ Y₁, A(X₁, Y₁) ← B(X₁),
σ₄ : D(X₄, Y₄) ← A(Z₄, X₄) \land A(Z₄, Y₄).
\]

It is not shy because frontier variables X₂ and Y₃ in σ₂ are both attacked by the existential variable Y₁ in σ₁, which makes the two body atoms in σ₂ chained. Yet the ontology is separable, as one can verify for each AQ q, rewₜ(Σₜₑₙₚₚ) is a Datalog rewriting of Σₜₑₙₚₚ.

### 4.3 The Block Finite Class

The class of separable ontologies still cannot cover some useful Datalog rewritable cases. Note that Σₑₙₚₚ in Example 4 is not separable, yet for each AQ, a Datalog rewriting does exist. It can be seen as follows, one can obtain Σ’ with the following rule to replace σₕ:

\[
σₕ’ : C(X₃) ← B(X₃, Y₃).
\]

Note that Σₑₙₚₚ \models σₕ’ and hence Σₑₙₚₚ and Σ’ are semantically equivalent (i.e., having the same models); and Σ’ is separable. Hence, for any CQ q, rewₜ(Σ’ \cup \{q\}) is a Datalog rewriting of Σₑₙₚₚₚ \cup \{q\}.

To define a Datalog rewritable class that includes Σₑₙₚₚ, we adapt sep-rewriting to allow blocks of, instead of individual, body atoms to be separately rewritten. For an ontology Σ and a rule σ, a block B of σ w.r.t. Σ is a minimal non-empty set of body atoms, i.e., B ⊆ body(σ), such that for each atom α in B, all the body atoms chained to α are also in B. Each rule with a non-empty body contains at least one block.

We associate each block B from the initial OMQ or generated during the rewriting with a fresh predicate P_B, called a block predicate. For a Datalog rule ρ, its block separation, denoted bsep(ρ), consists of the following Datalog rules

\[
\text{head}(ρ) ← \bigwedge \{B \land P_B(X_B) \} \land \text{body}(ρ).
\]

\[
P_B(X_B) ← \bigwedge \{B \} \land \text{body}(ρ).
\]
where \( X_B = (X, \cup V_{\text{body}(\rho)} B) \cap B \). Also, we reuse the block predicates in the same way as for separating predicates, by assigning \( \lambda(P_B) = (B, X_B) \).

The separate rewriting (or bsep-rewriting) sequence of an OMQ \( \Sigma_q \) is a sequence of Datalog queries \( \Pi'_b (i \geq 0, \text{with the same query predicate}) \), where \( \Pi'_b = \text{bsep}(q) \) and

\[
\Pi'_{b+1} = \text{cover}(\Pi'_b \cup \bigcup_{\sigma \in \Sigma, \rho \in \Pi'_b} \text{bsep}(\sigma(\rho, \mu)))
\]

for \( i > 0 \) with separating and block predicates being reused whenever possible. The bsep-rewriting of \( \Sigma_q \), denoted \( \text{rew}_b(\Sigma_q) \), is \( \Pi'_b \) for the smallest \( k \geq 0 \) such that \( \Pi'_{b+1} \) is equivalent to \( \Pi'_b \) or \( \Pi'_k \) if such a \( k \) does not exist.

Unlike \( \text{rew}_s(\Sigma_q) \), \( \text{rew}_b(\Sigma_q) \) may not be finite, but whenever it is so, it is a Datalog rewriting.

**Proposition 4.** For an OMQ \( \Sigma_q \), \( \text{rew}_b(\Sigma_q) \) is a Datalog rewriting of \( \Sigma_q \) whenever \( \text{rew}_b(\Sigma_q) \) is finite.

**Proof.** Let \( \Pi = \text{rew}_b(\Sigma_q) \). For each dataset \( D \), we want to show \( \Pi(D) = \Sigma_q(D) \). To show \( \Pi(D) \subseteq \Sigma_q(D) \), we can use a simple induction to show \( \Pi(D) \subseteq \Pi(D) \) in a similar way as in the proof of Lemma 1. In particular, the reuse of block predicates does not change the derivation of any new facts over \( \mathcal{R} \), as clearly if \( P_B \) is reused to replace \( P_B' \), then \( \lambda(P_B) \) is equivalent to \( \lambda(P_B') \) up to variable renaming and the blocks they respectively represent would have same instantiations on the universal models.

To show \( \Sigma_q(D) \subseteq \Pi(D) \), similar as in the proof of Theorem 1, we want to show that for each unfolding path \( \theta \) of \( \Sigma_q \), its resulting CQ can be obtained by unfolding Datalog rules \( \theta_1, \ldots, \theta_n \) in \( \Pi \), where \( \theta_i \) is of the form (3) or (4) for \( 1 \leq i \leq n \). Again, we can show this by an induction on the length of \( \theta \). In particular, suppose \( \theta \) can be rewritten by a TGD \( \sigma \) with piece unifier \( \mu = \langle B, H, \tau \rangle \), then there is a partition of \( B = B_1 \cup \cdots \cup B_m \) with \( B_i \cap B_j = \emptyset \) for \( 1 \leq i < j \leq m \), such that \( B_i \) can be separately resolved in the rewriting by \( \sigma \); that is, formally, \( \theta \mu \sigma(\tau) \) is equivalent to \( \theta \mu_1 \sigma_1(\tau_1) \cdots \mu_m \sigma_m(\tau_m) \) where \( \mu_i = \langle B_i, H_i, \tau_i \rangle \) for \( 1 \leq i \leq m \), with each \( H_i \subseteq H \) and \( \tau = \tau_1 \cdots \tau_m \). For simplicity, we assume \( B \) is one of such \( B_i \), and cannot be further partitioned. Then, by the definition of bsep(\( \rho \)), \( B \) is contained in the body of \( \theta_\rho \) (after substitution) for some \( 1 \leq k \leq n \) that is of the form (4). That is, \( \theta_k \) can be rewritten by \( \sigma \). Hence, following a similar process as in the proof of Theorem 1, we can show that \( \theta \mu \sigma(\tau) \) can be obtained from unfolding Datalog rules in \( \Pi \).

We define a new class of ontologies for which the result of bsep-rewriting is finite for any CQ and produces a Datalog rewriting.

**Definition 4.** An ontology \( \Sigma \) is block finite if for each CQ \( q \), \( \text{rew}_b(\Sigma_q) \) is finite.

From the definition, an OMQ with a block finite ontology always admits a Datalog rewriting. The class of block finite ontologies is denoted as BFin.

**Example 6.** The ontology \( \Sigma_{bc} \) in Example 4 is block finite, as for any CQ \( q \), its bsep-rewriting generates finitely many blocks.

On the other hand, \( \Sigma_{se} \cup \Sigma_{ir} \) is not block finite. As for a CQ \( Q = \langle A(X, Y) \rangle \), its bsep-rewriting generates infinitely many blocks of the form \( \{A(X, Z_1), A(Z_1, Z_2), \ldots, A(Z_n, Y) \} \) with an unbounded \( n \geq 0 \).

Next, we identify a concrete subclass of BFin that can be verified syntactically. To define the set of TGDs that may apply in the rewriting of a block, we recall the notion of rule dependency from [2]. An instance \( I \) depends on a TGD \( \sigma \) if there exist an instance \( I' \), a homomorphism \( \pi \) from body(\( \sigma \)) to \( I' \) that substitute existential variables in \( \sigma \) with fresh nulls (not in \( I \cup I' \)) and a homomorphism \( \pi' \) from \( I \) to \( I' \) head(\( \sigma \)) but not to \( I' \) (i.e., \( I \subseteq I' \)). For an ontology \( \Sigma \), the dependent module of a block \( B \) in \( \Sigma \), denoted \( \Sigma_B \), is the minimal set of TGDs \( \sigma \in \Sigma \) such that \( B \) depends on \( \sigma \) or some TGD in \( \Sigma_B \) contains a block that depends on \( \sigma \).

**Definition 5.** An ontology \( \Sigma \) is block expandable if for each block \( B \) in \( \Sigma \) consisting of more than one atoms, \( \Sigma_B \) belongs to Lin \( \cup \text{Stky} \cup \text{aGRD} \).

Intuitively, if an ontology \( \Sigma \) is block expandable, then for each block \( B \) consisting more than one atoms, the number of blocks generated via expanding \( B \) (by applying the TGDs in \( \Sigma_B \)) is finite.

**Example 7.** The ontology \( \Sigma_{be} \) in Example 4 does not belong to Lin \( \cup \text{Stky} \cup \text{aGRD} \), but it is block expandable. The only block consisting more than one atoms is \( B = \{A(X_1, Y_1), A(Y_1, Z_2) \} \) in \( \sigma_3 \), and \( \Sigma_B = \{\sigma_1, \sigma_2\} \), which belongs to Lin.

On the other hand, \( \Sigma_{bc} \cup \Sigma_{ir} \) is not block expandable. This is because \( \Sigma_B \) now include the transitivity rule, making it non-FUS.

The class of block expandable ontologies is denoted BExp, and we can show that each block expandable ontology is block finite.

**Theorem 2.** Lin \( \cup \text{Stky} \cup \text{aGRD} \cup \text{Shy} \subseteq \text{BExp} \subseteq \text{BFin} \).

To prove Theorem 2, we need the following lemma.

**Lemma 3.** For a block expandable ontology \( \Sigma \) and a CQ \( q \), a finite number of blocks is generated (up to variable renaming) in the unfolding sequence of \( \Sigma_q \).

**Proof of Theorem 2.** For the first set containment, suppose an ontology \( \Sigma \) belongs to Lin \( \cup \text{Stky} \cup \text{aGRD} \), then for each block \( B \) in \( \Sigma, \Sigma_B \subseteq \Sigma \) and also belongs to Lin \( \cup \text{Stky} \cup \text{aGRD} \). Hence, \( \Sigma \) belongs to BExp. Suppose \( \Sigma \) belongs to Shy, then \( \Sigma \) does not contain any block consisting of more than one atoms, and again \( \Sigma \) belongs to BExp. The strictness of the set containment has been shown in Examples 4 and 7, where \( \Sigma_{bc} \) belongs to BExp but is neither Sep, hence not Shy, nor Lin \( \cup \text{Stky} \cup \text{aGRD} \).

For the second set containment, suppose \( \Sigma \) belongs to BExp, we want to show it belongs to BFin. For a CQ \( q \), by the definition of bsep-rewriting sequence, each block in \( \Sigma \cup \{q\} \) generated during the rewriting is put in the body of a rule of the form (4) and is further rewritten. The blocks generated in the bsep-rewriting of \( \Sigma_q \) are included in (indeed, the same as) those generated in its unfolding. Note that a block cannot contain block predicates, which would violate the definition of a block. By Lemma 3, a finite number of blocks are generated (up to variable renaming), and due to the reuse of block predicates, there are a finite number of block predicates. Hence, the number of Datalog rules of the form (4) is finite. To see the number of Datalog
rules of the form (3) is also finite, let \( n \) be the maximum number of variables occurring in a TGD in \( \Sigma \) or a block generated during the bsep-rewriting of \( \Sigma_q \), and \( n \) is a finite number. Then, in the bsep-rewriting sequence, for each \( \sigma(\rho, \mu) \), where \( \sigma \in \Sigma \) and body(\( \rho \)) is a single block, it has at most 2l variables. Hence, the number of Datalog rules of the form (3) is bounded by the number of block predicates and the number of variables in it.

The strictness of the second set containment can be seen from Example 8.

**Example 8.** An example of a block finite ontology that is not block expandable is \( \Sigma = \Sigma_{\text{sep}} \cup \Sigma_{\text{tr}} \) with \( \Sigma_{\text{tr}} \) from Example 4 and \( \Sigma_{\text{sep}} \) from Example 5.

The ontology does not belong to BExp because there is a block \( B = \{ A(Z_4, X_4), A(Z_4, Y_4) \} \) in \( \sigma_4 \) (noting that \( \{ A(X_0, Z_0), A(Z_0, Y_0) \} \) in \( \sigma_0 \) is not block), and \( \Sigma_B = \{ \sigma_0, \sigma_1 \} \) is not in Lin \( \cup \) Stky \( \cup \) aGRD. Yet \( \Sigma \) belongs to BFin, as for any CQ \( q \), its bsep-rewriting generates finitely many blocks.

Finally, the classes Sep and BFin do not contain each other. We have seen an example \( \Sigma_{\text{sep}} \) that is in BFin but not in Sep. The following ontology is in Sep but not in BFin.

**Example 9.** The ontology with the following TGDs is separable but not block finite:

\[
\begin{align*}
\sigma_1 : & \exists Y_1. A(X_1, Y_1) \leftarrow B(X_1), \\
\sigma_2 : & B(X_2) \leftarrow A(Y_2, X_2), \\
\sigma_3 : & C(X_3) \leftarrow A(X_3, Y_3) \land C(Y_3).
\end{align*}
\]

It is separable, as each individual atom can be rewritten separately. Yet, it is not block finite, as the bsep-rewriting of a CQ \( Q \leftarrow C(X) \) generates infinitely many blocks of the form \( \{ A(X, Y_1), A(Y_1, Y_2), \ldots, A(Y_{n-1}, Y_n), C(Y_n) \} \) with an unbounded \( n \geq 0 \).

### 5 Implementation and Evaluations

We have implemented a prototype system, DREWer (Datalog REWriting for Existential Rules), and conducted two sets of experiments to evaluate the performance of our system. The system and benchmarks used in our evaluations can be found at https://www.ict.griffith.edu.au/aist/Drewer/.

#### 5.1 Implementation

Our system DREWer implements the bsep-rewriting, as well as checkers to verify if an ontology belongs to a specific class, such as Shy and BExp, as described in the previous section. In particular, to identify chained atoms and blocks in a given ontology, the method performs a forward propagation to mark variables in the TGDs that are attacked by existential variables. The ontology belongs to Shy if its atoms are not chained. To check whether the ontology belongs to BExp, a block dependency graph is constructed with the nodes of the graph being the identified blocks in the ontology. Each block \( B \) is associated with the set of TGDs it depends on, and there is an edge from \( B \) to a block \( B' \) if \( B \) depends on a TGD \( \sigma \) that contains \( B' \). By traversing the block dependency graph, the dependent module of a block can be efficiently computed and its membership in Lin, Shy, or aGRD can be verified.

For the rewriting process, the piece unification module is adapted from the first-order rewriting system Graal\(^1\) [21]. As an optimisation, our method also implements the query pre-processing as in [20]. Intuitively, for an OMQ \( \Sigma_q \), we first eliminate from CQ \( q \) those atoms that can be inferred by other atoms in \( q \) together with ontology \( \Sigma \). Such an optimisation is particularly effective for ontologies in Lin or those with a large number of linear TGDs.

The bsep-rewriting process separates each TGDs, including those generated during the rewriting, according to the blocks in them. As the body of each separated TGD consists of only one block, the resulting TGDs are much more compact than the first-order rewritings, which also effectively simplified the search for piece unifications (for TGD unfolding). To identify blocks in the generated TGDs, from the proof of Lemma 3, new blocks are formed only by combining existing blocks. Hence, it is possible to pre-compute all the potential blocks that can be generated during the rewriting. Also, as we observed, the reuse of block predicates is rare in practice whereas the checking of reuse conditions can be expensive. Hence, for efficiency, predicate reuse is switched off as long as the termination of rewriting is guaranteed.

#### 5.2 Benchmark Ontologies

We adopted several commonly used ontologies for ontology-mediated query answering [13], [17], [19], [21], [34]. These benchmark OMQs consist of ontologies with a diversity of expressiveness, queries with different complexity, and data with various sizes.

Table 1 provides a summary of the benchmark OMQs in terms of their expressiveness (Exp), the numbers of rules (#R), the number of facts (#F), and the ontology classes they belong to. These classes include Lin, Stky, aGRD, Shy, and BExp. Note that the class BExp is new and introduced in Section 4. Besides the aforementioned classes, we also considered the class of weakly acyclic ontologies [32], denoted WA, which guarantees chase termination.

The ontologies in these OMQs fall into three groups. Those in the first group, obtained from [17], [19], [21], are in major DL-Lite variants, and thus are first-order rewritable. In particular, RS [19] has a simple ontology but specially crafted long queries (with up to 15 atoms), which is a known challenge to existing rewriting-based systems. The ontologies in the second group are widely used OWL2 ontologies mostly obtained from the PAGoDA benchmarks [13]. As some OWL2 axioms cannot be directly translated into TGDs, such as class unions and universal property restrictions, we used the OWL2 parser provided by Graal to convert an OWL ontology into a set of TGDs in the format of DLGP supported by Graal. The ontologies in the third group are benchmarks for chase-based systems from ChaseBench [34]. These ontologies allow TGDs with predicates of arity more than two and thus may not be in OWL. In particular, the DEEP ontology is designed as a “pure stress” benchmark in the sense that it aims to cause the chase procedure to dive into deep chase chains (which roughly correspond to long rewriting sequences). It comes with three versions DEEP100, DEEP200, and DEEP300 of increasing numbers of TGDs.

5.3 Evaluations on Query Rewriting

For the evaluations on query rewriting, we compared our system Drewer with state-of-the-art query rewriting systems. Graal is one of few first-order rewriting systems for general TGDs, while Rapid\(^2\) [17] and Grind\(^1\) [18] are Datalog rewriting systems for description logic ontologies. In particular, Rapid generates Datalog rewritings using an optimized resolution-based method and moreover, can compute the first-order rewritings by unfolding the Datalog rules. Also, Iqaros\(^4\) [14] is a first-order rewriting system for OWL2 ontologies featuring in its rewriting minimisation. Nyaya [20], another first-order rewriting system that can handle TGDs, is not included in our evaluations, as Graal is reported to outperform Nyaya [21].

In our first experiment, for each ontology, we evaluated five queries that come with the benchmark to compare the size of rewriting and time efficiency for these systems (except for RS which has only three long queries). The experimental results are shown in Table 2. The sizes are measured by the numbers of rules and the times are in milliseconds. Rapid(D) and Rapid(U) denote respectively the Datalog and UCQ rewritings for Rapid. We set a 5 minutes time limit per query, ran each query 3 times, and reported the average. The time limit is based on the performance of major baseline systems compared in our experiments and as we will see, Drewer takes only a few miliseconds or a few seconds for most OMQs. “TO” denotes time out whereas a “-” means the system could not handle the OMQ (or reported errors).

To achieve efficiency and compactness in rewriting, Graal makes use of the so-called compiled pre-orders to replace certain (linear Datalog) rules in the ontologies. For example, the Datalog rule $A(x) \leftarrow B(x)$ is replaced by a preorder $B \preceq A$, which is not considered in rewriting but can be directly used in query answering. Hence, the UCQ rewritings produced by Graal are of small sizes, but they need to be coupled with the compiled pre-orders for query answering. For a fair comparison, when we measure the rewriting sizes, we used the format $x+y$ for Graal, where $x$ is the number of CQs in the UCQ rewriting and $y$ is the number of Datalog rules corresponding to the compiled pre-orders. Since not all pre-orders are used for specific queries, we tracked for each query those pre-orders actually used for query answering (following Graal’s native query answering process) and $y$ is the number of only those used pre-orders.

It can be seen from Table 2 that the sizes of Datalog rewritings are often comparable to or much smaller than those of UCQ rewritings. While Drewer sometimes produces larger rewritings than other systems, the differences are marginal; and the increase in sizes may result from Drewer splitting up rules and introducing fresh predicates. On the other hand, unlike UCQ rewritings whose sizes may easily grow exponentially when the query gets long (e.g., the three long queries of RS), the sizes of Datalog rewritings produced by Drewer depend on the numbers of blocks. In cases where a UCQ rewriting turns out to be large, the numbers of blocks can still be relatively small. And Drewer is also superior or comparable to other systems regarding the time efficiency of rewriting. This can be seen especially from OBOProtein, RS, and DEEP300, which are generally considered challenging benchmarks for rewriting. In particular, all the other systems timed out on q5 for OBOProtein, whereas Drewer took only less than a second to complete the rewriting. And all the other systems except for Graal failed to complete their rewriting on the three long queries for RS, due to the large sizes of rewritings. While Graal’s optimisation works well for most benchmarks, it is less effective on expressive ontologies like DEEP300. The reason

---

2. www.image.ece.ntua.gr/~gstam/
3. www.informatik.uni-bremen.de/tdki/hansen/doku.php?id=grind
4. code.google.com/p/iqaros/
for Graal’s timeouts on Reactome and Uniprot is that its algorithm cannot terminate on those queries, due to certain TGDs that are not first-order rewritable. Although Rapid and Iqaros could successfully complete their rewriting on Reactome and Uniprot, they are dedicated to lightweight description logics such as DL-Lite and $\mathcal{EL}$, which means they might ignore or simplify some expressive TGDs.

In our second experiment, we evaluated the scalability of our rewriting system with respect to query complexity. Apart from the queries provided with the benchmarks, we also implemented a query generator to generate CQs with specified lengths (i.e., number of atoms). Our generator randomly selects predicates from the ontology and ensures that atoms in each query can be unified with TGDs in the ontology. For each ontology, we generated 10 queries for that atoms in each query can be unified with TGDs in the ontology. Our generator also implemented a query generator to generate CQs with specified lengths (i.e., number of atoms). Our generator randomly selects predicates from the ontology and ensures that atoms in each query can be unified with TGDs in the ontology. For each ontology, we generated 10 queries for each length for various lengths between 1 and 20, and we compared the scalability of Drewer and Graal over queries of increasing lengths. Figure 1 shows the results for five ontologies, expressed both in complex TGDs (DEEP100, STB-128 and ONT-256) and in lightweight description logics (Adolena and LUBM).

It can be seen that for ontologies with complex TGDs, Drewer significantly outperforms Graal regarding both time efficiency and rewriting sizes. Also, our system scales better as query length increases. For DEEP100, Graal shows exponential blow up and could not handle queries with lengths more than 4. Due to the compiled pre-orders, Graal is rather effective in processing ontologies in lightweight description logics and the increase in the rewriting sizes is rather mild when the queries get longer, yet the rewriting time increases rather significantly. On the other hand, while the sizes of our Datalog rewriting are similar to those of Graal, our system again demonstrates better efficiency and scalability in rewriting, especially as the queries get longer; indeed, the

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Query</th>
<th>Drewer Rapid(D)</th>
<th>Grind</th>
<th>Gaal Rapid(U)</th>
<th>Iqaros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adolena</td>
<td>q1</td>
<td>43 8</td>
<td>43 9</td>
<td>27 367</td>
<td>2+29 22</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>32 8</td>
<td>31 7</td>
<td>28 363</td>
<td>2+29 21</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>34 8</td>
<td>32 8</td>
<td>27 362</td>
<td>1+31 19</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>37 25</td>
<td>39 9</td>
<td>33 367</td>
<td>2+34 24</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>40 73</td>
<td>37 9</td>
<td>32 378</td>
<td>1+36 21</td>
</tr>
<tr>
<td></td>
<td>q6</td>
<td>3 5</td>
<td>4 4</td>
<td>2 324</td>
<td>1+1 13</td>
</tr>
<tr>
<td></td>
<td>q7</td>
<td>1 4</td>
<td>46 10</td>
<td>38 364</td>
<td>1+0 10</td>
</tr>
<tr>
<td></td>
<td>q8</td>
<td>4 3</td>
<td>20 6</td>
<td>20 355</td>
<td>1+47 13</td>
</tr>
<tr>
<td></td>
<td>q9</td>
<td>3 4</td>
<td>63 13</td>
<td>55 337</td>
<td>1+1 13</td>
</tr>
<tr>
<td></td>
<td>q10</td>
<td>8 5</td>
<td>52 12</td>
<td>42 357</td>
<td>1+6 13</td>
</tr>
<tr>
<td>LUMB</td>
<td>q1</td>
<td>1420 89</td>
<td>1276 70</td>
<td>1152 17977</td>
<td>1+1275 186</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>96 990</td>
<td>92 33</td>
<td>26 172773</td>
<td>5+87 200</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>192 64</td>
<td>155 14</td>
<td>147 182875</td>
<td>1+154 101</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>73 1660</td>
<td>81 33</td>
<td>37 178808</td>
<td>19+62 186</td>
</tr>
<tr>
<td>OpenGALEN2</td>
<td>q1</td>
<td>30 63</td>
<td>29 11</td>
<td>229971</td>
<td>20+7 156</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>1360 637</td>
<td>1356 998</td>
<td>1358 23237</td>
<td>1264+92 7088</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>34742 185</td>
<td>33919 7514</td>
<td>1+3918 344</td>
<td>33887 564</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>34805 1961</td>
<td>34879 84318</td>
<td>- -</td>
<td>34733 1102</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>1389 668</td>
<td>27907 TO</td>
<td>- -</td>
<td>TO TO 36612</td>
</tr>
<tr>
<td>RS</td>
<td>q1</td>
<td>16 127</td>
<td>TO TO  TO</td>
<td>TO TO</td>
<td>14+4 561</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>22 977</td>
<td>TO TO  TO</td>
<td>TO TO</td>
<td>100+4 16387</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>23 10756</td>
<td>TO TO  TO</td>
<td>TO TO</td>
<td>143+4 46557</td>
</tr>
<tr>
<td>Reactome</td>
<td>q1</td>
<td>3 3</td>
<td>3 6</td>
<td>- -</td>
<td>1+2 34</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>16 4</td>
<td>13 6</td>
<td>- -</td>
<td>TO TO 32</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>16 5</td>
<td>13 6</td>
<td>- -</td>
<td>TO TO 32</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>16 5</td>
<td>15 7</td>
<td>- -</td>
<td>TO TO 32</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>16 4</td>
<td>15 6</td>
<td>- -</td>
<td>TO TO 32</td>
</tr>
<tr>
<td>Uniprot</td>
<td>q1</td>
<td>1 4</td>
<td>1 3</td>
<td>- -</td>
<td>1+0 15</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>1 4</td>
<td>1 3</td>
<td>- -</td>
<td>1+0 15</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>1 4</td>
<td>12 6</td>
<td>- -</td>
<td>1+0 19</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>15 2</td>
<td>11 6</td>
<td>- -</td>
<td>TO TO 11</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>24 8</td>
<td>12 6</td>
<td>- -</td>
<td>TO TO 11</td>
</tr>
<tr>
<td>DEEP300</td>
<td>q1</td>
<td>25 13</td>
<td>- -</td>
<td>- -</td>
<td>117+2 971</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>30 15</td>
<td>- -</td>
<td>- -</td>
<td>528+3 13072</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>30 14</td>
<td>- -</td>
<td>- -</td>
<td>624+2 17485</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>234 55</td>
<td>- -</td>
<td>- -</td>
<td>TO TO</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>55 23</td>
<td>- -</td>
<td>- -</td>
<td>TO TO</td>
</tr>
<tr>
<td>STB-128</td>
<td>q1</td>
<td>8 7</td>
<td>- -</td>
<td>- -</td>
<td>- 4+2 25</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>7 6</td>
<td>- -</td>
<td>- -</td>
<td>4+2 52</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>7 6</td>
<td>- -</td>
<td>- -</td>
<td>8+0 46</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>14 6</td>
<td>- -</td>
<td>- -</td>
<td>16+5 176</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>12 7</td>
<td>- -</td>
<td>- -</td>
<td>8+5 60</td>
</tr>
<tr>
<td>ONT-256</td>
<td>q1</td>
<td>11 8</td>
<td>- -</td>
<td>- -</td>
<td>8+2 56</td>
</tr>
<tr>
<td></td>
<td>q2</td>
<td>16 7</td>
<td>- -</td>
<td>- -</td>
<td>4+6 64</td>
</tr>
<tr>
<td></td>
<td>q3</td>
<td>14 8</td>
<td>- -</td>
<td>- -</td>
<td>8+6 84</td>
</tr>
<tr>
<td></td>
<td>q4</td>
<td>16 8</td>
<td>- -</td>
<td>- -</td>
<td>12+5 108</td>
</tr>
<tr>
<td></td>
<td>q5</td>
<td>16 8</td>
<td>- -</td>
<td>- -</td>
<td>48+1 2410</td>
</tr>
</tbody>
</table>

Table 2: Comparison on query rewriting
rewriting times of our system almost remain constant as the lengths of the queries increase.

In our third experiment, we evaluated the success rates of rewriting systems in handling large ontologies. We selected the two ontologies OpenGALEN2 and OBOprotein, each containing more than 20K TGDs. For each of these two ontologies, we generated 30 queries of various lengths, ranging from 2 to 20. Figure 2 shows the success rates of various query rewriting systems on these two ontologies, where the rewriting of a query is successful if it can be completed within 5 minutes.

The advantage of Datalog rewriting is clearly demonstrated in handling large ontologies, as the Datalog rewriting systems Drewer and Rapid could successfully handle all the queries for both ontologies while the first-order rewriting systems failed on most of the queries (especially all the queries for both ontologies while the first-order rewriting systems Drewer and Rapid could successfully handle all the queries for both ontologies while the first-order rewriting systems failed on most of the queries (especially for OpenGALEN2). Our system is still more efficient than Rapid, spending an average of 900 milliseconds per query, compared to Rapid spending averagely 6 seconds per query.

5.4 Evaluation on Query Answering

We have demonstrated the advantages of our Datalog rewriting in regarding to the sizes and time efficiency, but one may wonder if the smaller Datalog rewritings indeed lead to more efficient query answering compared to first-order rewritings. In the second set of experiments, we evaluate the benefits of our Datalog rewriting with respect to query answering. In particular, we evaluated the time efficiency of query answering based on different rewritings (and query-related modules) over increasing data sizes, ontology sizes, and query lengths.

To evaluate query answering on increasing data sizes, we used the LUBM benchmark which comes with a data generator. We generated data of various sizes, ranging from 13M to 55M, and used the five original queries that come with the benchmark. To evaluate query answering on increasing ontology sizes, we used the DEEP benchmark, with three versions containing 1.1K, 1.2K and 1.3K TGDs respectively. Again, we used the five original queries that come with the DEEP benchmark. Finally, to evaluate query answering on increasing query lengths, we used the ONT-256 benchmark which has 2M data. We used our query generator to generate queries of lengths from 1 to 14.

To compare our Datalog rewriting with first-order rewriting produced by Graal, we used a Datalog translation for both of its UCQ rewritings and the used pre-orders, and deployed the same Datalog engine VLog5 for query answering. Note that VLog is more efficient than Graal’s native query answering engine. In [10], the authors show a goal-oriented approach would significantly improve the efficiency of a chase-based query answering system. Yet we were unable to obtain their system for our comparison. We implemented a naïve method to extract a module of an ontology that is relevant to a given query, by computing the dependant module of the query in the ontology (which is defined by generalising the dependant module of a block in Section 4.3). We used the query-related modules as a baseline to evaluate the benefits of various rewritings in query answering.

Figure 3 shows the times (averaged over the queries) of query answering based on our Datalog rewritings, the Datalog translation of Graal’s first-order rewritings, and the extracted query-related modules. The times recorded are those for Vlog to perform chase on the Datalog rules (from rewritings or modules) and to answer the query over the chase, and excludes the times for producing the rewriting (which have been evaluated before in experiments on rewriting time efficiency) or for module extraction.

Overall, rewritings have clear time efficiency benefits compared to query-related modules, which separates the benefits of rewriting (as a goal-oriented procedure in contrast to a chase procedure) in eliminating irrelevant TGDs. And our Datalog rewritings can be processed more efficiently than first-order rewritings from Graal, especially on large ontologies like DEEP and long queries. In particular, our Datalog rewritings show better scalability with respect to the increasing query lengths.

6 Conclusion

In this paper, we have presented a novel approach for compact Datalog rewriting of CQs and TGDs. In particular, we separate the body atoms in the rewriting results to avoid long rules and reuse the introduced predicates for termination. We have also introduced several new Datalog rewritable classes based on variants of our Datalog rewriting methods. In particular, the block expandable class is a concrete class whose membership can be verified syntactically, and it contains several well known classes. Furthermore, we have implemented and evaluated our system, Drewer, which is capable to handle a wide range of practical ontologies. Our system showed superior performance compared to state-of-the-art rewriting and query answering systems.

For future work, we are working on identifying more concrete classes of Datalog rewritable TGDs, optimising our rewriting algorithm and implementation, and validating the approach in some practical applications. We are also interested in exploring parallel or distributed Datalog engines, such as SociaLite [35] and BigDatalog [36], for efficient query answering with our Datalog rewriting.

Acknowledgements

We would like to thank the three anonymous referees for their helpful comments. This work was partially supported by the National Natural Science Foundation of China under grant 61976153.

References

Fig. 1: Comparison on the rewriting of queries with different lengths

Fig. 2: Success rates of rewriting on large ontologies

Fig. 3: Comparison on query answering with different rewritings


Zhe Wang is a senior lecturer at Griffith University, Australia. His research interests include ontology-based data access, semantic technologies, and knowledge representation and reasoning. His research has led to 60 publications in influential journals and conferences in AI.

Peng Xiao has just completed his PhD program on ontology-based data access in existential rules at Griffith University. His research interests include rule-based systems and query answering in ontologies.

Kewen Wang is a professor at Griffith University, Australia. His current research interests include reasoning and learning in knowledge-based systems. He has been publishing in top conferences and journals in AI and is currently an Area Editor of the Journal of Web Semantics.

Zhiqiang Zhuang is a professor at the College of Intelligence and Computing, Tianjin university, China. His research interests include belief revision, knowledge representation and reasoning, and social network analysis. He has regularly published in top AI journals and conferences such as AAAI and IJCAI.

Hai Wan is an associate professor at Sun Yat-Sen University, China. His research interests are in knowledge graphs and reasoning.