Query Rewriting and Optimization for Ontological Databases

GEORG GOTTLOB, GIORGIO ORSI, and ANDREAS PIERIS, University of Oxford

Ontological queries are evaluated against a knowledge base consisting of an extensional database and an ontology (i.e., a set of logical assertions and constraints that derive new intensional knowledge from the extensional database), rather than directly on the extensional database. The evaluation and optimization of such queries is an intriguing new problem for database research. In this article, we discuss two important aspects of this problem: query rewriting and query optimization. Query rewriting consists of the compilation of an ontological query into an equivalent first-order query against the underlying extensional database. We present a novel query rewriting algorithm for rather general types of ontological constraints that is well suited for practical implementations. In particular, we show how a conjunctive query against a knowledge base, expressed using linear and sticky existential rules, that is, members of the recently introduced Datalog$^{\pm}$ family of ontology languages, can be compiled into a union of conjunctive queries (UCQ) against the underlying database. Ontological query optimization, in this context, attempts to improve this rewriting process soas to produce possibly small and cost-effective UCQ rewritings for an input query.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing, rule-based databases, relational databases*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Inference engines, logic programming, resolution*

General Terms: Algorithms, Theory, Languages, Performance

Additional Key Words and Phrases: Ontological query answering, tuple-generating dependencies, query rewriting, query optimization

ACM Reference Format:
Georg Gottlob, Giorgio Orsi, and Andreas Pieris. 2014. Query rewriting and optimization for ontological databases. ACM Trans. Datab. Syst. 39, 3, Article 25 (September 2014), 46 pages.
DOI: http://dx.doi.org/10.1145/2638546

1. INTRODUCTION

1.1. Ontological Database Management Systems

The use of ontological reasoning in companies, governmental organizations, and other enterprises has become widespread in recent years. An ontology is an explicit specification of a conceptualization of an area of interest and consists of a formal representation of knowledge as a set of concepts within a domain, as well as the relationships between instances of these concepts. Moreover, ontologies have been adopted as high-level conceptual descriptions of the data contained in data repositories that are sometimes distributed and heterogeneous in the data models. Due to their high expressive power, ontologies are also replacing more traditional conceptual models such as UML class diagrams and Entity Relationship schemata.

This research has received funding from the ERC grant 246858 "DIADEM". G. Orsi also acknowledges the Oxford Martin School's Institute for the Future of Computing grant LC0910-019. A. Pieris also acknowledges the EPSRC grant EP/J00846/1 "PrOQAW".
Authors' addresses: G. Gottlob, G. Orsi (corresponding author), and A. Pieris, Department of Computer Science, University of Oxford, Wellington Square, Oxford OX1 2JD, UK; email: giorgio.orsi@cs.ox.ac.uk; A. Pieris, Department of Computer Science, University of Oxford, Wellington Square, Oxford OX1 2JD, UK.

DOI: http://dx.doi.org/10.1145/2638546

We are currently witnessing the marriage of ontological reasoning and database technology, giving rise to a new type of database management systems, the so-called ontological database management systems, equipped with advanced reasoning and query processing mechanisms [Calvanese et al. 2007; Calì et al. 2011]. More precisely, an extensional database D is combined with an ontology Σ that is used to derive new intensional knowledge from the extensional database. An input conjunctive query is not just answered against the database as in the classical setting, but against the logical theory (a.k.a. ontological database) $D \cup \Sigma$—recall that conjunctive queries correspond to the select-project-join fragment of relational algebra and form one of the most natural and commonly used languages for querying relational databases [Abiteboul et al. 1995]. Therefore, the answer to a conjunctive query $\exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$ with distinguished variables $\mathbf{X}$ over the ontological database consists of all tuples $\mathbf{t}$ of constants such that, when we substitute the variables $\mathbf{X}$ with $\mathbf{t}$, $\exists \mathbf{Y} \varphi(\mathbf{t}, \mathbf{Y})$ evaluates to *true* in every model of $D \cup \Sigma$, that is, in every instance containing D and satisfying Σ.

This amalgamation of different technologies stems from the need for semantically enhancing existing databases with ontological constraints. Indeed, database technology providers have recognized this need and have recently started to build ontological reasoning modules on top of their existing software with the aim of delivering effective database management solutions to their customers. For example, Oracle, Inc. offers a system, called Oracle Database 11g, enhanced by modules performing ontological reasoning tasks[1]. Also, Ontotext offers a family of semantic repositories, called OWLIM[2], and Semafora Systems develops an inference machine, called Ontobroker[3], for processing ontologies that support all of the World Wide Web Consortium (W3C) recommendations. Enhancing databases with ontologies is also at the heart of several research-based systems such as QuOnto [Acciarri et al. 2005] and Quest [Rodriguez-Muro and Calvanese 2012].

1.2. Ontology Languages

Ontologies are modeled using formal languages called ontology languages. Description Logics (DLs) [Baader et al. 2003] are a family of knowledge representation languages widely used in ontological modeling. In fact, DLs model a domain of interest in terms of concepts and roles that represent classes of individuals and binary relations on classes of individuals, respectively. Interestingly, DLs provide the logical underpinning for the Web Ontology Language (OWL) and its revision OWL 2, as standardized by the W3C[4]. Unfortunately, in order to achieve favorable computational properties, DLs are able only to describe knowledge for which the underlying relational structure is treelike. Moreover, they usually allow only unary and binary relations. The overcoming of the aforesaid limitations through the definition of expressive rule-based ontology languages has become in the last years a field of intense research in the KR and database communities. In fact, traditional database constraints such as *tuple-generating dependencies* (TGDs) (a.k.a. *existential rules* and *Datalog*$^{\pm}$ *rules*) of the form $\forall \mathbf{X} \forall \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z}\, \psi(\mathbf{X}, \mathbf{Z})$, where φ and ψ are conjunctions of atoms over a relational schema, appeared to be a suitable formalism for ontological modeling and reasoning. Examples of such languages can be found in Baget et al. [2011], Krötzsch and Rudolph [2011], and Calì et al. [2012a, 2012b].

A desirable property of an ontology language, apart from ensuring the decidability, is to guarantee the tractability of conjunctive query answering w.r.t. the *data complexity*,

[1]http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html.
[2]http://www.ontotext.com/owlim.
[3]http://www.semafora-systems.com/en/products/ontobroker/.
[4]http://www.w3.org/TR/owl2-overview/.

that is, the complexity calculated by considering only the database as part of the input. Indeed, the data complexity of query answering is widely regarded as more meaningful and relevant in practice than the *combined complexity* (calculated by considering everything as part of the input), since the query and ontology are typically of a size that can be productively assumed fixed and usually are much smaller than a typical relational database. Several lightweight DLs have been proposed that guarantee that conjunctive query answering is feasible in polynomial time w.r.t. the data complexity. Such DLs are $\mathcal{EL}$ [Baader 2003] and the members of the *DL-Lite* family [Calvanese et al. 2007; Poggi et al. 2008], namely DL-Lite$_\mathcal{R}$, DL-Lite$_\mathcal{F}$, and DL-Lite$_\mathcal{A}$. These languages can be seen as tractable sublanguages of OWL; in fact, the language DL-Lite$_\mathcal{R}$ forms the OWL 2 QL[5] profile of OWL 2. It was convincingly argued that, despite their simplicity, $\mathcal{EL}$ and the DL-Lite formalisms are powerful enough for modeling an overwhelming number of real-life scenarios. More recently, several classes of TGDs have been identified that guarantee the same low data complexity for conjunctive query answering. For example, the class of *guarded* TGDs (inspired by the guarded fragment of first-order logic [Andréka et al. 1998]) that is noticeably more general than $\mathcal{EL}$ and the members of the DL-Lite family has been investigated in Calì et al. [2008]. Extensions of guarded TGDs can be found in Baget et al. [2011] and Krötzsch and Rudolph [2011]. Moreover, the classes of *linear* and *sticky* TGDs, both encompassing the DL-Lite family, have been proposed in Calì et al. [2012a, 2012b].

1.3. First-Order Rewritability

Polynomial-time tractability is often considered not good enough for efficient query processing. Ideally, one would like to achieve the same complexity as for processing first-order queries, or, equivalently, (nonrecursive) SQL queries. An ontology language $\mathcal{L}$ guarantees the *first-order rewritability* of conjunctive query answering if, for every conjunctive query q and ontology Σ expressed in $\mathcal{L}$, a positive first-order query q_Σ called *perfect rewriting*[6] can be constructed such that, given a database D, q_Σ evaluated over D yields exactly the same result as q evaluated against the ontological database $D \cup \Sigma$ [Calvanese et al. 2007]. Since answering first-order queries is in AC_0 in data complexity [Vardi 1995], it immediately follows that query answering under ontology languages that guarantee the first-order rewritability of the problem is also in AC_0 in data complexity.

First-order rewritability is a most desirable property since it ensures that the query answering process can be largely decoupled from data access. In fact, as depicted in Figure 1, to answer a query q over an ontological database $D \cup \Sigma$, a separate software can compile q into q_Σ, then translate q_Σ into a standard SQL query $q^\star$, and finally submit it to the underlying relational database management system holding D, where it is evaluated and optimized in the usual way.

Example 1.1. Consider the set Σ consisting of the TGD:

$$\forall X \forall Y \, \mathit{project}(X), \mathit{inArea}(X, Y) \rightarrow \exists Z \, \mathit{hasCollaborator}(Z, Y, X),$$

asserting that each project has an external collaborator specialized in the area of the project. We can ask for projects in the area of databases for which there are external collaborators by posing the CQ $\exists A \, \mathit{hasCollaborator}(A, db, B)$. Intuitively, due to the preceding TGD, not only do we have to query *hasCollaborator*, but we also need to look for projects in the area of databases, as such projects will necessarily have an external

[5]http://www.w3.org/TR/owl2-profiles/.

[6]In general, there exist more than one perfect rewritings. However, for query answering, all the possible rewritings are equivalent and thus we can refer to *the* perfect rewriting.

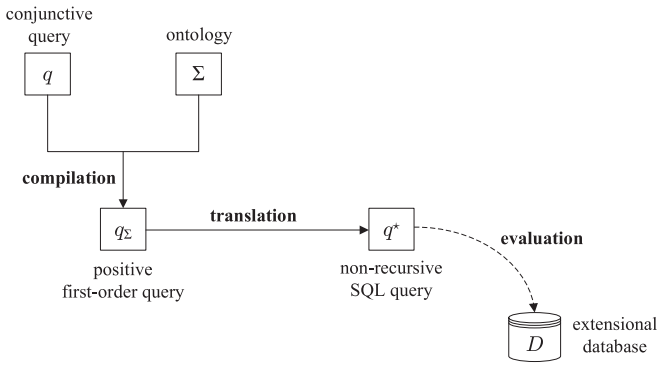

Fig. 1. Answering queries via rewriting.

```
SELECT C.p_id FROM hasCollaborator C
WHERE C.area = 'db'
UNION
SELECT P.p_id FROM project P, inArea A
WHERE A.area = 'db' AND P.p_id = A.p_id.
```

Fig. 2. The SQL query of Example 1.1.

collaborator. The perfect rewriting q_Σ will thus be the union of CQs:

$$(\exists A\, hasCollaborator(A, db, B)) \ \vee \ (project(B) \wedge inArea(B, db)).$$

Assuming the schema $project(\mathsf{p\_id})$, $inArea(\mathsf{p\_id}, \mathsf{area})$, $hasCollaborator(\mathsf{c\_id}, \mathsf{area}, \mathsf{p\_id})$, it is clear that q_Σ can be written in SQL as shown in Figure 2.

Interestingly, the members of the DL-Lite family of DLs, as well as the classes of linear and sticky TGDs, guarantee the first-order rewritability of conjunctive query answering. Actually, the previously named languages guarantee a stronger property than first-order rewritability: given a conjunctive query q and an ontology Σ expressed in one of the aforesaid formalisms, the perfect rewriting q_Σ can be expressed as a union of conjunctive queries, that is, we do not need the full expressive power of positive first-order queries. As we explain shortly, the main problem that we address in this article is precisely the question of how to compute q_Σ correctly and efficiently when the input ontology Σ is expressed as a set of linear or sticky TGDs.

1.4. Aims and Objectives

The advantage of first-order rewritability is obvious, that is, conjunctive query answering can be deferred to a standard query language such as SQL, in turn allowing to exploit mature and efficient existing database technology that is accessible via the underlying database management system. However, there is a drawback in this approach in that, if the algorithm that constructs the perfect rewriting inflates the query excessively and creates from a reasonably sized ontological query a massive exponentially sized SQL query, then even the best database management system may be of little use. This problem gave rise to flourishing research activity in the DL community. A remarkable number of rewriting algorithms, with the aim of compiling a conjunctive query and a DL-Lite ontology into a "small" union of conjunctive queries, have been proposed in the last five years (see, e.g., Calvanese et al. [2007], Pérez-Urbina et al. [2010], Chortaras et al. [2011], Kikot et al. [2012a], and Venetis et al. [2013]); see Section 2.

Surprisingly, before the conference version of the present article [Gottlob et al. 2011], no practical algorithm, able to efficiently compile a conjunctive query and an ontology modeled using an expressive TGD-based language into a union of conjunctive queries, was available. It is the precise aim of this work to fill this gap for linear and sticky TGDs. Both linearity and stickiness are well-accepted paradigms.

—A TGD is called *linear* if it has only one body atom [Calì et al. 2012a]; notice that the body is the left-hand side of the implication. Despite its simplicity, linearity forms a robust language with several applications. Linear TGDs are strictly more expressive than the description logic DL-Lite$_{\mathcal{R}}$ [Calvanese et al. 2007] that, as already said, forms the OWL 2 QL profile of W3C's standard ontology language for modeling Semantic Web ontologies. Importantly, linear TGDs, in contrast to DL-Lite$_{\mathcal{R}}$, can be used with relational database schemas of arbitrary arity. The usefulness of schemas of higher arity (not just unary and binary relations) has been recognized by the DL community; as evidence we mention DLR-Lite [Calvanese et al. 2013a], a recent generalization of DL-Lite to arbitrary arity, which is also captured by linear TGDs. Also, linear TGDs generalize inclusion dependencies, a well-known class of relational constraints; in fact, inclusion dependencies can be equivalently written as TGDs with just one body atom and one head atom without repeated variables. Moreover, linear TGDs are powerful enough to express conditional inclusion dependencies that extend traditional inclusion dependencies by enforcing bindings of semantically related data values. They are useful in data cleaning and contextual schema mapping [Bohannon et al. 2006; Bravo et al. 2007]. In fact, conditional inclusion dependencies can be written as linear TGDs with constant values in the body. Furthermore, linear TGDs generalize local-as-view (LAV) TGDs that are employed in data exchange and data integration to define schema mappings, that is, specifications that describe how data for a source schema can be transformed into data for a target schema; see, for example, ten Cate and Kolaitis [2009]. Finally, linear TGDs can be used in schema evolution and in particular for expressing the decompose operator, with the aim of splitting a table into smaller tables [Curino et al. 2013].

—Stickiness [Calì et al. 2012b] allows joins to appear in rule bodies not expressible via linear TGDs, let alone via DL(R)-Lite assertions; more details are given in Section 3. Interestingly, sticky TGDs are able to capture well-known data modeling constructs such as (conditional) inclusion and multivalued dependencies. Furthermore, sticky TGDs, in contrast to linear TGDs (and most of the existing DLs), allow to describe knowledge for which the underlying relational structure is not treelike. This is mainly due to the fact that sticky TGDs are expressive enough for encoding the cartesian product of two tables. For instance, the set of sticky TGDs consisting of $\forall X \forall Y \; p_i(X, Y) \rightarrow \exists Z \, p_i(Y, Z), s_i(Z)$, for each $i \in \{1, 2\}$, and $\forall X \forall Y \; s_1(X), s_2(Y) \rightarrow r(X, Y)$, computes the cartesian product of s_1 and s_2 that forms an infinite clique, thus the underlying relational structure has infinite treewidth. As already observed by the DL community, there are some natural ontological statements, such as "all elephants are bigger than all mice" [Rudolph et al. 2008], that are expressible only via cartesian product assertions. Notice that the preceding statement can be captured by the sticky TGD $\forall X \forall Y \; \mathit{elephant}(X), \mathit{mouse}(Y) \rightarrow \mathit{biggerThan}(X, Y)$. Finally, sticky TGDs can also be used for schema evolution purposes and in particular for expressing the merge operator, with the aim of putting together two or more tables [Curino et al. 2013].

Apart from designing a practical rewriting algorithm for linear and sticky TGDs, we would also like to investigate the possibility of improving the computation of the perfect rewriting on multicore architectures commonly available in modern database servers. In the long term, we envision relational database systems able to handle ontological

constraints natively, as done today for traditional data dependencies such as primary and foreign keys. A key difference is that ontological constraints are not supposed to be enforced by the DBMS as classical integrity constraints, but rather to be taken into consideration during the evaluation of a query. This article is a significant step towards this direction.

1.5. The Existing Approach

Although it is known that both linear and sticky TGDs guarantee the first-order rewritability of conjunctive query answering, the existing algorithms are of theoretical nature and it is generally accepted that there is no obvious way how they will lead to better practical rewriting algorithms. The key property of linear and sticky TGDs that implies the first-order rewritability of conjunctive query answering is the so-called *bounded derivation-depth property* (BDDP) [Calì et al. 2012a]. As we shall see in Section 3, to compute the answer to a conjunctive query q over an ontological database $D \cup \Sigma$, where Σ is a linear or sticky ontology, it suffices to evaluate q over a special model of $D \cup \Sigma$ that can be homomorphically embedded into every other model of $D \cup \Sigma$. Such a model, called *universal model* (a.k.a. *canonical model*), always exists and can be constructed by applying the *chase procedure*, a powerful tool for reasoning about data dependencies. Intuitively, the chase adds new atoms to the extensional database D, possibly involving null values that act as witnesses for the existentially quantified variables until the final result, denoted $chase(D, \Sigma)$, satisfies Σ. However, $chase(D, \Sigma)$ is in general infinite and thus not explicitly computable. The BDDP implies that it suffices to evaluate q over an initial finite part of $chase(D, \Sigma)$ that depends only on q and Σ. Roughly, $chase(D, \Sigma)$ can be decomposed into levels, where database atoms have level zero while an inferred atom has level $k + 1$ if it is obtained due to atoms with maximum level k; we refer to the part of the chase up to level k as $chase^k(D, \Sigma)$. Thus, the BDDP implies that there exists $k \geqslant 0$ such that, for every database D, the answer to q over $D \cup \Sigma$ coincides with the answer to q over $chase^k(D, \Sigma)$. An algorithm for computing the prefect rewriting q_Σ by exploiting the preceding property has been presented in Calì et al. [2012a]. Roughly, one can enumerate all the possible database ancestors $D_1, \ldots, D_n$ of the image of the given query and then, starting from each D_i, construct $chase^k(D, \Sigma)$, where k is the depth provided by the BDDP, that will then give rise to a query in the final rewriting. It is evident that such a procedure is computationally expensive and also that the obtained queries are usually very large and cannot be effectively materialized. Notice that the goal of Calì et al. [2012a] was to establish that classes of TGDs which enjoy the BDDP guarantee the first-order rewritability of conjunctive query answering, without taking into account implementation issues. It is apparent that we had to look for new rewriting procedures that substantially deviate from the one described earlier.

1.6. Summary of Contributions

Our contributions can be summarized as follows.

(1) We propose a novel query rewriting algorithm, called XRewrite, that is based on backward chaining resolution. In fact, XRewrite uses the TGDs as rewriting rules, with the aim of simulating, independently from the extensional database, the chase derivations responsible for the generation of the image of the input query. Such an algorithm is better for practical applications than the one described before since, during the rewriting process, we only explore the part of the chase that is needed in order to entail the query, namely the proof of the query, thus we avoid the generation of a nonnegligible number of useless atoms. Interestingly, XRewrite is sound and complete even if we consider an arbitrary set of TGDs without any

syntactic restrictions; however, in this general case the termination of the algorithm is not guaranteed. We show that, if the input set of TGDs is linear or sticky, then XRewrite terminates, and thus it forms a practical query rewriting algorithm for linear and sticky TGDs; recall that the design of such an algorithm is the main research challenge of this work.

(2) We present a parallel version of XRewrite, called XRewriteParallel, with the aim of reducing the overall execution time for computing the final rewriting by exploiting multicore architectures. To the best of our knowledge, this is the first attempt to design a parallel query rewriting algorithm. The key idea is to decompose the input query q into smaller queries $q_1, \ldots, q_m$, where $m \geqslant 1$, in such a way that each q_i can be rewritten independently by concurrent rewriters into a query Q_{q_i} and then merge the queries $Q_{q_1}, \ldots, Q_{q_m}$ in order to obtain the final rewriting.

(3) We propose a technique, called query elimination, aiming at optimizing the final rewritten query under linear TGDs. Query elimination, which is an additional step during the execution of XRewrite, reduces: *(i)* the size of the final rewriting, *(ii)* the number of atoms in each query of the rewriting, and *(iii)* the number of joins to be executed. The key idea underlying query elimination is that the linearity of TGDs allows to effectively identify atoms in the body of a query that are logically implied (w.r.t. a given set of TGDs) by other atoms in the same query.

(4) After implementing our algorithm, we have analyzed its behavior and identified certain operations, such as the computation of the most general unifier for a set of atoms, that might benefit from caching. We also perform an extensive analysis on the impact of our optimizations on the rewriting process and show that all of them reduce the number of redundant queries in the final rewriting. We finally compare our system with ALASKA (i.e., the reference implementation of König et al. [2012]), the only known system supporting ontological query rewriting under arbitrary TGDs. We observe that both systems return minimal rewritings on the given test cases. However, query elimination allows us to perform a better exploration of the rewriting search space on most of the given test cases. Interestingly, even for those cases where ALASKA performs a better exploration of the search space, our algorithm achieves better performance due to the caching mechanism. Notably, on certain test cases, the parallelization of the rewriting provides a fundamental contribution towards making the rewriting manageable as the number of explored and generated queries is drastically reduced.

Roadmap. After a review of previous work on query rewriting in Section 2 and some technical definitions and preliminaries in Section 3, we proceed with our new results. In Section 4, we present the rewriting algorithm XRewrite and in Section 5 its parallel version. In Section 6, we present the query elimination technique. Implementation issues are discussed in Section 7, while the experimental evaluation is presented in Section 8. We conclude in Section 9 with a brief outlook on further research.

2. RELATED WORK ON QUERY REWRITING

An early query rewriting algorithm for the DL-Lite family of DLs, introduced in Calvanese et al. [2007] and implemented in the QuOnto system, reformulates the given query into a union of conjunctive queries. The size of the reformulated query is unnecessarily large. This is mainly due to the fact that the factorization step (that is needed, as we shall see, to guarantee completeness) is applied in a "blind" way, even if not needed, and as a result many superfluous queries are generated. In Pérez-Urbina et al. [2010] an alternative resolution-based rewriting algorithm for DL-Lite$_{\mathcal{R}}$ is proposed and implemented in the Requiem system that addresses the issue of the useless factorizations (and therefore of the redundant queries generated due to this

weakness) by directly handling existential quantification through proper functional terms; notice that this algorithm works also for more expressive DLs that do not guarantee first-order rewritability of query answering (in this case, the computed rewriting is a recursive Datalog query). A query rewriting algorithm for DL-Lite$_{\mathcal{R}}$, called Rapid, that is more efficient than the one in Pérez-Urbina et al. [2010], is presented in Chortaras et al. [2011]. The efficiency of Rapid is based on the selective and stratified application of resolution rules; roughly, it takes advantage of the query structure and applies a restricted sequence of resolutions that may lead to useful and redundant-free rewritings. An alternative query rewriting technique for DL-Lite$_{\mathcal{R}}$ is presented in Kikot et al. [2012a]. Although the obtained rewritings are, in general, not correct and of exponential size, in most practical cases the rewritings are correct and of polynomial size. In Venetis et al. [2013], the problem of computing query rewritings for DL-Lite$_{\mathcal{R}}$ in an incremental way is investigated. More precisely, a technique that computes an extended query by "extending" a previously computed rewriting of the initial query (thus avoiding recomputation) is proposed.

The algorithms mentioned before leverage specificities of DLs, such as the limit to unary and binary predicates only and the absence of variable permutations in the axioms. Therefore, they cannot be easily extended to more general TGD-based languages; in fact, DL-based systems often resort to case-by-case analysis on the syntactic form of the DL axioms. Following a more general approach, the works Gottlob et al. [2011] and König et al. [2012, 2013] presented a backward-chaining rewriting algorithm able to deal with arbitrary TGDs, providing that the language under consideration satisfies suitable syntactic restrictions that guarantee the termination of the algorithm. In other works that follow a different approach, instead of computing a union of conjunctive queries, the rewritings are expressed in some other query language such as nonrecursive Datalog. These can be found in the literature [Rosati and Almatelli 2010; Orsi and Pieris 2011; Gottlob and Schwentick 2012; Kikot et al. 2012b; Thomazo 2013; Gottlob et al. 2014].

A related field is that of database query reformulation in presence of views and constraints [Deutsch et al. 1999; Halevy 2001; Benedikt et al. 2014]. Given a conjunctive query q and a set of constraints Σ, the goal is to find all the minimal equivalent reformulations of q w.r.t. Σ. The most widely used approach in this respect is the chase-and-backchase algorithm [Deutsch et al. 1999] implemented in the MARS system [Deutsch and Tannen 2003], discussed in more detail in Section 6.

3. DEFINITIONS AND BACKGROUND

3.1. Technical Definitions

We present background material necessary for this article. We recall some basics on relational databases, relational queries, tuple-generating dependencies, and the chase procedure relative to such dependencies. For further details on the previous notions we refer the reader to Abiteboul et al. [1995].

Alphabets. We define the following pairwise disjoint (countably infinite) sets of symbols: a set Γ of *constants* (constituting the "normal" domain of a database), a set Γ_N of *labeled nulls* (used as placeholders for unknown values and thus that can also be seen as globally existentially quantified variables), and a set Γ_V of (regular) *variables* (used in queries and dependencies). Different constants represent different values (*unique name assumption*), while different nulls may represent the same value. A fixed lexicographic order is assumed on $\Gamma \cup \Gamma_N$ such that every value in Γ_N follows all those in Γ. We denote by $\mathbf{X}$ sequences (or sets, with a slight abuse of notation) of variables $X_1, \ldots, X_k$, with $k \geqslant 1$. Throughout, let $[n] = \{1, \ldots, n\}$, for any integer $n \geqslant 1$.

Relational Model. A *relational schema* $\mathcal{R}$ (or simply *schema*) is a set of *relational symbols* (or *predicates*), each with its associated arity. We write r/n to denote that the predicate r has arity n. By $arity(\mathcal{R})$ we refer to the maximum arity over all predicates of $\mathcal{R}$. A *position* $r[i]$ (in $\mathcal{R}$) is identified by a predicate $r \in \mathcal{R}$ and its i-th argument (or attribute). A *term* t is a constant, null, or variable. An *atomic formula* (or simply *atom*) has the form $r(t_1, \ldots, t_n)$, where r/n is a relation and $t_1, \ldots, t_n$ are terms. For an atom $\underline{a}$, we denote by $terms(\underline{a})$ and $var(\underline{a})$ the set of its terms and the set of its variables, respectively. These notations naturally extend to sets of atoms. Conjunctions of atoms are often identified with the sets of their atoms. An *instance* I for a schema $\mathcal{R}$ is a (possibly infinite) set of atoms of the form $r(\mathbf{t})$, where $r/n \in \mathcal{R}$ and $\mathbf{t} \in (\Gamma \cup \Gamma_N)^n$. A *database* D is a finite instance such that $terms(D) \subset \Gamma$.

Substitutions. A *substitution* from a set of symbols S to a set of symbols S' is a function $h : S \to S'$ defined as follows: $\varnothing$ is a substitution (empty substitution) and, if h is a substitution, then $h \cup \{t \to t'\}$ is a substitution, where $t \in S$ and $t' \in S'$; if $t \to t' \in h$, then we write $h(t) = t'$. An assertion of the form $t \to t'$ is called *mapping*. The *restriction* of h to $T \subseteq S$, denoted $h|_T$, is the substitution $h' = \{t \to h(t) \mid t \in T\}$. A *homomorphism* from a set of atoms A to a set of atoms A' is a substitution $h : \Gamma \cup \Gamma_N \cup \Gamma_V \to \Gamma \cup \Gamma_N \cup \Gamma_V$ such that if $t \in \Gamma$, then $h(t) = t$ and if $r(t_1, \ldots, t_n) \in A$, then $h(r(t_1, \ldots, t_n)) = r(h(t_1), \ldots, h(t_n)) \in A'$. A set of atoms $A = \{\underline{a}_1, \ldots, \underline{a}_n\}$, where $n \geqslant 2$, *unifies* if there exists a substitution γ, called *unifier* for A, such that, $\gamma(\underline{a}_1) = \cdots = \gamma(\underline{a}_n)$. A *most general unifier* (MGU) for A is a unifier for A, denoted as γ_A, such that, for each other unifier γ for A, there exists an substitution γ' such that $\gamma = \gamma' \circ \gamma_A$. Notice that if a set of atoms unify, then there exists an MGU. Furthermore, the MGU for a set of atoms is unique (modulo variable renaming).

Datalog. A *Datalog rule* ρ is an expression of the form $\underline{a}_0 \leftarrow \underline{a}_1, \ldots, \underline{a}_n$, for $n \geqslant 0$, where $\underline{a}_i$ is an atom containing constants of Γ and variables of Γ_V and where every variable occurring in $\underline{a}_0$ must appear in at least one of the atoms $\underline{a}_1, \ldots, \underline{a}_n$; the latter is known as the safety condition. The atom $\underline{a}_0$ is called the *head* of ρ, denoted as $head(\rho)$, while the set of atoms $\{\underline{a}_1, \ldots, \underline{a}_n\}$ is called the *body* of ρ, denoted as $body(\rho)$. A *Datalog program* Π over a schema $\mathcal{R}$ is a set of Datalog rules such that, for each $\rho \in \Pi$, the predicate of $head(\rho)$ does not occur in $\mathcal{R}$. The program Π is *nonrecursive* if there is some ordering $\rho_1, \ldots, \rho_n$ of the rules of Π so that the predicate in the head of ρ_i does not occur in the body of a rule ρ_j, for each $j \leqslant i$. The *extensional database* (EDB) predicates are those that do not occur in the head of any rule of Π; all the other predicates are called *intensional database* (IDB) predicates. A *model* of Π is an instance I for $\mathcal{R}$ such that, for every Datalog rule of the form $\underline{a}_0 \leftarrow \underline{a}_1, \ldots, \underline{a}_n$ appearing in Π, I satisfies the first-order formula $\forall \mathbf{X}(\underline{a}_1 \wedge \ldots \wedge \underline{a}_n \rightarrow \underline{a}_0)$, where $\mathbf{X}$ are the variables occurring in ρ, in other words, whenever there exists a homomorphism h such that $h(\{\underline{a}_1, \ldots, \underline{a}_n\}) \subseteq I$, $h(\underline{a}_0) \in I$. The semantics of Π w.r.t. a database D for $\mathcal{R}$, denoted as $\Pi(D)$, is the minimum model of Π containing D (which is unique and always exists).

Queries. An n-ary *Datalog query* Q over a schema $\mathcal{R}$ is a pair $\langle \Pi, p \rangle$, where Π is a Datalog program over $\mathcal{R}$ and p is an n-ary (output) predicate occuring in the head of at least one rule of Π. Q is a *nonrecursive Datalog query* if Π is nonrecursive. Q is a *union of conjunctive queries* (UCQs) if Π is nonrecursive, p is the only IDB predicate in Π, and, for each rule $\rho \in \Pi$, p does not occur in $body(\rho)$. Finally, Q is a *conjunctive query* (CQ) if it is a union of CQs and Π contains exactly one rule. The *answer* to an n-ary Datalog query $Q = \langle \Pi, p \rangle$ over a database D is the set $\{\mathbf{t} \in \Gamma^n \mid p(\mathbf{t}) \in \Pi(D)\}$, denoted $Q(D)$. Since the output predicate of a (U)CQ is clear from the syntax of the query, in the rest of the article, for brevity, a CQ is seen as a Datalog rule while a UCQ is seen as a Datalog program (instead of a pair consisting of a program and a predicate). The

variables occurring in the head of a CQ are its *distinguished variables*. The answer to a CQ q[7] over a (possibly infinite) instance I can be equivalently defined as the set of all tuples of constants $\mathbf{t}$ for which there exists a homomorphism h such that $h(body(q)) \subseteq I$ and $h(\mathbf{X}) = \mathbf{t}$, where $\mathbf{X}$ are the distinguished variables of q. The answer to a UCQ Q over I can be equivalently defined as the set of tuples $\{\mathbf{t} \mid$ where there exists $q \in Q$ such that $\mathbf{t} \in q(I)\}$.

Tuple-Generating Dependencies. A *tuple-generating dependency* (TGD) σ over a schema $\mathcal{R}$ is a first-order formula $\forall\mathbf{X}\forall\mathbf{Y}\,\varphi(\mathbf{X},\mathbf{Y}) \rightarrow \exists\mathbf{Z}\,\psi(\mathbf{X},\mathbf{Z})$, where $\mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z} \subset \Gamma_V$ and where φ, ψ are conjunctions of atoms over $\mathcal{R}$ (possibly with constants). Formula φ is the *body* of σ, denoted $body(\sigma)$, while ψ is the *head* of σ, denoted $head(\sigma)$. Henceforth, for brevity, we will omit the universal quantifiers in front of TGDs and use the comma (instead of $\wedge$) for conjoining atoms. Such σ is satisfied by an instance I for $\mathcal{R}$, written $I \models \sigma$, if the following holds: whenever there exists a homomorphism h such that $h(\varphi(\mathbf{X},\mathbf{Y})) \subseteq I$, then there exists a homomorphism $h' \supseteq h|_{\mathbf{X}}$, called the *extension* of $h|_{\mathbf{X}}$, such that $h'(\psi(\mathbf{X},\mathbf{Z})) \subseteq I$. An instance I satisfies a set Σ of TGDs, denoted $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$. A set Σ of TGDs is in *normal form* if each of its TGDs has a single head atom containing only one occurrence of an existentially quantified variable. As shown, for instance, in Calì et al. [2012b], every set Σ of TGDs over a schema $\mathcal{R}$ can be transformed in logarithmic space into a set $\mathsf{N}(\Sigma)$ over a schema $\mathcal{R}_{\mathsf{N}(\Sigma)}$ in normal form of size at most quadratic in $|\Sigma|$, such that Σ and $\mathsf{N}(\Sigma)$ are equivalent w.r.t. query answering; for more details see electronic Appendix A.1.

Conjunctive Query Answering under TGDs. Given a database D for a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$, the answers we consider are those that are true in all models of D w.r.t. Σ. Formally, the *models* of D w.r.t. Σ, denoted as $mods(D,\Sigma)$, is the set of all instances I such that $I \supseteq D$ and $I \models \Sigma$. The *answer* to an n-ary CQ q w.r.t. D and Σ, denoted as $ans(q, D, \Sigma)$, is the set of n-tuples $\{\mathbf{t} \mid \mathbf{t} \in q(I),$ for each $I \in mods(D,\Sigma)\}$; the answer to an n-ary UCQ is defined analogously. Notice that the associated decision problem, that asks whether a tuple of constants belongs to the answer of a CQ w.r.t. a database and a set of TGDs, is undecidable under arbitrary TGDs [Beeri and Vardi 1981]; in fact, it remains undecidable even when the schema and the set of TGDs are fixed [Calì et al. 2008], or even when the set of TGDs is a singleton [Baget et al. 2011]. Concrete classes of TGDs that are of special interest for the current work and also guarantee the decidability of query answering are presented in Section 3.3.

The TGD Chase Procedure. The *chase procedure* (or simply *chase*) is a fundamental algorithmic tool introduced for checking implication of dependencies [Maier et al. 1979], and later for checking query containment [Johnson and Klug 1984]. Informally, the chase is a process of repairing a database w.r.t. a set of dependencies so that the resulting instance satisfies the dependencies. By abuse of terminology, we shall use the term "chase" interchangeably for both the procedure and its result. The chase works on an instance through the so-called *TGD chase rule*.

> *TGD chase rule.* Consider an instance I for a schema $\mathcal{R}$, as well as a TGD $\sigma : \varphi(\mathbf{X},\mathbf{Y}) \rightarrow \exists\mathbf{Z}\,\psi(\mathbf{X},\mathbf{Z})$ over $\mathcal{R}$. We say that σ is *applicable* to I if there exists a homomorphism h such that $h(\varphi(\mathbf{X},\mathbf{Y})) \subseteq I$. The result of *applying* σ to I with h is $I' = I \cup h'(\psi(\mathbf{X},\mathbf{Z}))$ and we write $I\langle\sigma, h\rangle I'$, where h' is an extension of $h|_{\mathbf{X}}$ such that $h'(Z)$ is a "fresh" labeled null of Γ_N not occurring in I, and following lexicographically all those in I, for each $Z \in \mathbf{Z}$. In fact, $I\langle\sigma, h\rangle I'$ defines a single TGD chase step.

[7]Henceforth, for clarity, we usually use lower-case letters for CQs and upper-case letters for UCQs.

Let us now give the formal definition of the *chase* of a database w.r.t. a set of TGDs. A *chase sequence* of a database D w.r.t. a set Σ of TGDs is a sequence of chase steps $I_i\langle\sigma_i, h_i\rangle I_{i+1}$, where $i \geqslant 0$, $I_0 = D$ and $\sigma_i \in \Sigma$. The chase of D w.r.t. Σ, denoted $chase(D, \Sigma)$, is defined as follows.

—A *finite chase* of D w.r.t. Σ is a finite chase sequence $I_i\langle\sigma_i, h_i\rangle I_{i+1}$, where $0 \leqslant i < m$ and there is no $\sigma \in \Sigma$ applicable to I_m; let $chase(D, \Sigma) = I_m$.
—An infinite chase sequence $I_i\langle\sigma_i, h_i\rangle I_{i+1}$, where $i \geqslant 0$, is *fair* if, whenever a TGD $\sigma : \varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists\mathbf{Z}\,\psi(\mathbf{X}, \mathbf{Z})$ is applicable to I_i with homomorphism h, then there exists an extension h' of $h|_{\mathbf{X}}$ and $k > i$ such that $h'(head(\sigma)) \subseteq I_k$. An *infinite chase* of D w.r.t. Σ is a fair infinite chase sequence $I_i\langle\sigma_i, h_i\rangle I_{i+1}$, where $i \geqslant 0$; let $chase(D, \Sigma) = \bigcup_{i=0}^{\infty} I_i$.

Let $chase^{[k]}(D, \Sigma)$ be the instance constructed after $k \geqslant 0$ applications of the TGD chase step. An example of the chase procedure can be found in electronic Appendix A.1. It is well known that the chase of D w.r.t. Σ is a *universal model* of D w.r.t. Σ, that is, for each $I \in mods(D, \Sigma)$, there exists a homomorphism h_I such that $h_I(chase(D, \Sigma)) \subseteq I$ [Fagin et al. 2005; Deutsch et al. 2008]. Using this universality property, it can be shown that the chase is a formal algorithmic tool for query answering under TGDs. More precisely, the answer to a CQ q w.r.t. a database D and a set of TGDs Σ coincides with the answer to q over the chase of D w.r.t. Σ, namely $ans(q, D, \Sigma) = q(chase(D, \Sigma))$.

The TGD chase rule given before is known as *oblivious* since it "forgets" to check whether the TGD under consideration is already satisfied, that is, it adds atoms to the given instance even if it is not necessary. The version of the TGD chase rule that applies stricter criteria to the applicability of TGDs, with the aim of adding atoms to the given instance only if necessary, is called *restricted*. The universality property was originally shown for the restricted version of the chase [Fagin et al. 2005; Deutsch et al. 2008], which is considered as the standard one. However, as explicitly stated in Calì et al. [2013], the universality property holds also for the oblivious chase; this was established by showing the existence of a homomorphism from the oblivious to the restricted chase. Thus, for our purposes, we can safely consider the oblivious chase. This is done for technical clarity and simplicity. As discussed in Johnson and Klug [1984], even in the simple case of inclusion dependencies, things become technically more complicated if the restricted chase is employed, since the applicability of a TGD depends on the presence of other atoms previously constructed by the chase.

3.2. Query Answering via Rewriting

A fundamental property that a class of TGDs should enjoy is to guarantee the decidability of (the decision version) of conjunctive query answering; recall that in general this problem is undecidable. However, as already discussed in Section 1, to be able to work with very large datasets, decidability of query answering is not enough. We need also high tractability in data complexity, that is, when both the query and the set of TGDs are fixed and possibly feasible by the use of relational query processors. First-order rewritability, introduced in the context of description logics [Calvanese et al. 2007], guarantees the previous desirable properties. Roughly speaking, given a CQ and a set of TGDs, a (finite) first-order query can be constructed, called *perfect rewriting*, that takes into account the semantic consequences of the TGDs. Then, the answer to the input query w.r.t. a database D and the set of TGDs is obtained by evaluating the perfect rewriting directly over D. Formally, the problem of conjunctive query answering under a set of TGDs Σ is *first-order rewritable* if, for every CQ q, a (finite) positive first-order query q_Σ can be constructed such that, for every database D, $ans(q, D, \Sigma) = q_\Sigma(D)$. Unfortunately, the problem of deciding whether a set of TGDs guarantees the first-order rewritability of CQ answering is undecidable; see electronic Appendix A.2.

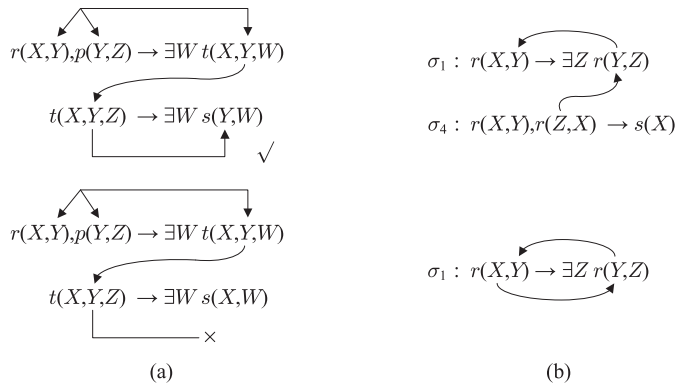

Fig. 3. Sticky property and propagation step.

It is well known that the evaluation of first-order queries is in the highly tractable class AC_0 in data complexity [Vardi 1995]. Recall that this is the complexity class of recognizing words in languages defined by constant-depth Boolean circuits with (unlimited fanin) AND and OR gates (see, e.g., Papadimitriou [1994]). Consequently, CQ answering under sets of TGDs that guarantee the first-order rewritability of the problem is in AC_0 in data complexity. Given that every first-order query can be equivalently written in (nonrecursive) SQL, in practical terms this means that CQ answering can be deferred to a standard query language such as SQL. This allows us to exploit all the optimization capabilities of the underlying RDBMS.

3.3. Concrete Classes of TGDs

Since the problem of identifying first-order rewritability is undecidable, it is not possible to syntactically characterize the fragment of TGDs that guarantees the first-order rewritability of CQ answering. However, several sufficient syntactic conditions have been proposed, the two main ones being linearity and stickiness.

Linearity. Linear TGDs have been proposed in Calì et al. [2012a]. A TGD σ is called *linear* if σ has only one body atom. The class of linear TGDs, namely the set of all possible sets of linear TGDs is denoted LINEAR. Despite its simplicity, as already discussed in Section 1.4, LINEAR is quite natural with several applications. Linear TGDs guarantee the first-order rewritability of CQ answering [Calì et al. 2012a]; this is also implicit in Baget et al. [2011], where atomic hypothesis rules that coincide with linear TGDs are investigated. This result was established by showing that LINEAR enjoys the BDDP. However, as already remarked in Section 1, the techniques based on the BDDP do not lead to practical query rewriting algorithms.

Stickiness. The class of sticky sets of TGDs, denoted STICKY, has been proposed in Calì et al. [2012b] with the aim of identifying an expressive class that allows for meaningful joins in rule bodies. The key idea underlying stickiness is to ensure that, during the chase, terms associated with body variables that appear more than once (i.e., join variables) are always propagated (or "stick") to the inferred atoms. This is illustrated in Figure 3(a).

The formal definition of sticky sets of TGDs hinges on a variable marking procedure called SMarking. This procedure accepts as input a set Σ of TGDs and returns the same set after marking some of its body variables. For notational convenience, given a TGD σ, an atom $\underline{a} \in \mathit{head}(\sigma)$, and a universally quantified variable V of σ, $\mathit{pos}(\sigma, \underline{a}, V)$ is the set of positions in $\underline{a}$ at which V occurs. SMarking(Σ) is constructed as follows.

First, we apply on Σ the *initial marking* step: for each $\sigma \in \Sigma$ and for each variable $V \in var(body(\sigma))$, if there exists an atom $\underline{a} \in head(\sigma)$ such that $V \notin var(\underline{a})$, then each occurrence of V in $body(\sigma)$ is marked. $\mathsf{SMarking}(\Sigma)$ is obtained by applying exhaustively (i.e., until a fixpoint is reached) on Σ the *propagation* step: for each pair $\langle \sigma, \sigma' \rangle \in \Sigma \times \Sigma$, for each atom $\underline{a} \in head(\sigma)$ and for each universally quantified variable $V \in var(\underline{a})$, if there exists an atom $\underline{b} \in body(\sigma')$ in which a marked variable occurs at each position of $pos(\sigma, \underline{a}, V)$, then each occurrence of V in $body(\sigma)$ is marked.

Example 3.1. Consider the set Σ consisting of

$$\sigma_1 : r(X, Y) \rightarrow \exists Z\, r(Y, Z) \quad \sigma_3 : s(X), s(Y) \rightarrow p(X, Y)$$
$$\sigma_2 : r(X, Y) \rightarrow s(X) \quad \sigma_4 : r(X, Y), r(Z, X) \rightarrow s(X).$$

By applying the initial marking step, the body variables of Σ are marked with a cap (i.e., $\hat{V}$) and, due to the propagation step, are marked with a double-cap as follows:

$$\sigma_1 : r(\hat{X}, \hat{\hat{Y}}) \rightarrow \exists Z\, r(Y, Z) \quad \sigma_3 : s(X), s(Y) \rightarrow p(X, Y)$$
$$\sigma_2 : r(X, \hat{Y}) \rightarrow s(X) \quad \sigma_4 : r(X, \hat{Y}), r(\hat{Z}, X) \rightarrow s(X).$$

Figure 3(b) depicts the two ways of propagating the marking to the variable Y of σ_1.

A set Σ of TGDs is called *sticky* if, for every $\sigma \in \mathsf{SMarking}(\Sigma)$, each marked variable appears only once. Stickiness guarantees the first-order rewritability of CQ answering [Calì et al. 2012a]. As for linear TGDs, this was established by showing that the BDDP holds and hence all the drawbacks of this approach are inherited.

Normal Form. Notice that the normalization procedure for TGDs, presented in electronic Appendix A.1, preserves linearity and stickiness. In other words, given a linear (respectively, sticky) set Σ of TGDs, the set $\mathsf{N}(\Sigma)$ is linear (respectively, sticky). Thus, in the rest of the article we assume without loss of generality that TGDs have only one head atom with at most one existentially quantified variable that occurs once. This assumption will allow us to simplify our later technical definitions and proofs. Given a TGD σ, we refer to the position of the (single) existentially quantified variable by $\pi_\exists(\sigma)$; if there is no existentially quantified variable, then $\pi_\exists(\sigma) = \varepsilon$.

4. UCQ REWRITING

In this section, we tackle the problem of CQ answering under linear and sticky sets of TGDs. Our goal is to design a rewriting algorithm well suited for practical applications. In particular, we present a backward-chaining rewriting algorithm that constructs a union of conjunctive queries. Let us say that our techniques apply immediately even if we additionally consider a limited form of functional dependencies and negative constraints of the form $\forall \mathbf{X}\, \varphi(\mathbf{X}) \rightarrow \bot$, where φ is a conjunction of atoms. Notice that these modeling features are vital for ontological reasoning purposes. Due to space reasons, we omit the details and refer the reader to electronic Appendix B.1.

4.1. An Informal Description

Given a CQ q and a set Σ of TGDs, the actual computation of the rewriting is done by exhaustively applying a backward resolution-based step, called *rewriting step*, that uses the rules of Σ as rewriting rules whose direction is right-to-left. More precisely, a rewriting step is applied on a CQ, starting from the given query q, and gives rise to a new CQ that will be part of the final rewriting. Intuitively, a rewriting step simulates, in the reverse direction (hence the term "backward"), an application of a TGD during the construction of the chase. In other words, by applying the rewriting step, we bypass an application of a TGD during the chase and the obtained query is one level closer to

the database level. This is done until there are no other TGD chase steps to bypass, meaning that we reached the database level, as required.

Example 4.1 (*Rewriting Step*). Consider the TGD and CQ given in Example 1.1 (are also given here):

$$\sigma \ : \ project(X), inArea(X, Y) \ \rightarrow \ \exists Z\, hasCollaborator(Z, Y, X),$$
$$q \ : \ p(B) \ \leftarrow hasCollaborator(A, db, B).$$

Observe that $head(\sigma)$ and $body(q)$ unify and that $\gamma = \{X \rightarrow B, Y \rightarrow db, Z \rightarrow A\}$ is their MGU. This intuitively means that an atom of the form $hasCollaborator(t_1, db, t_2)$, where t_1 and t_2 are terms, to which $body(q)$ can be homomorphically mapped, may be obtained during the construction of the chase by applying σ. Such a TGD chase step can be simulated (or bypassed) by applying the rewriting step on q using σ. This consists of replacing $body(q)$[8] with $body(\sigma)$ and then applying γ on the obtained query. The result of such a rewriting step is the CQ

$$q' \ : \ p(B) \ \leftarrow project(B), inArea(B, db),$$

and the final rewriting of q w.r.t. $\{\sigma\}$ is the UCQ $\{q, q'\}$.

The fact that a set $S \subseteq body(q)$ unifies with $head(\sigma)$ indicates that an atom $\underline{a}$, to which S can be homomorphically mapped, may be obtained during the chase by applying σ. However, this is not always true and may lead to erroneous rewriting steps that in turn will generate unsound rewritings. Let us illustrate the two cases via a simple example where the blind application of the rewriting step, without checking whether further conditions are satisfied, leads to unsound rewritings.

Example 4.2 (*Unsound Rewritings*). Consider the same TGD σ as in Example 4.1 and the CQ

$$q_1 \ : \ p(B) \ \leftarrow hasCollaborator(c, db, B),$$

where $c \in \Gamma$. Since $head(\sigma)$ and $body(q_1)$ unify, with $\gamma = \{X \rightarrow B, Y \rightarrow db, Z \rightarrow c\}$ being their MGU, we proceed with the rewriting step. This will result in the CQ:

$$q' \ : \ p(B) \ \leftarrow project(B), inArea(B, db).$$

Consider now the database $D = \{project(a), inArea(a, db)\}$. The CQ q' maps to D and we conclude that $\langle a \rangle \in q'(D)$. However, the original query q_1 does not map to $chase(D, \{\sigma\})$, since there is no atom of the form $hasCollaborator(c, db, t)$ in $chase(D, \{\sigma\})$ and thus $ans(q_1, D, \{\sigma\}) = \varnothing$. Therefore, no rewriting containing q' is a sound rewriting of q_1 with respect to $\{\sigma\}$. This is because the constant c is associated with the existentially quantified variable Z and thus, after applying the rewriting step, the information about the constant c occurring in the original query is lost.

Consider now the CQ

$$q_2 \ : \ p(B) \ \leftarrow hasCollaborator(B, db, B).$$

As before, $head(\sigma)$ and $body(q)$ unify and $\gamma = \{X \rightarrow B, Y \rightarrow db, Z \rightarrow B\}$ is their MGU. After applying the rewriting step we get again the CQ q', and $\langle a \rangle \in q'(D)$. However, there is no atom of the form $hasCollaborator(t, db, t)$, that is, an atom where the same term occurs at the first and the last position, meaning that $ans(q_2, D, \{\sigma\}) = \varnothing$. Hence, no rewriting containing q' is a sound rewriting of q_2 w.r.t. $\{\sigma\}$. This is because one occurrence of the variable B that is in a self-join, namely that occurs more than once

[8]Notice that, in general, only a subset of $body(q)$, that unifies with the head of the TGD under consideration, is replaced during a rewriting step.

in $body(q)$, is associated with the existentially quantified variable Z and hence, after applying the rewriting step, the fact that the variable B is in a self-join is lost.

The blind application of the rewriting step may also cause the generation of unsafe queries, that is, queries where a distinguished variable does not occur in the body. This may happen if a distinguished variable of the query to be rewritten is associated with an existentially quantified variable of the TGD under consideration. From the preceding informal discussion we conclude that the rewriting step can be applied on a set $S \subseteq body(q)$ using a TGD σ (or simply, σ is applicable to S) if the following hold: (1) S and $head(\sigma)$ unify; and (2) their MGU does not associate the constants, the join variables, and the distinguished variables of q with the existentially quantified variable of σ. This is the so-called *applicability condition* and its formal definition will be given in the next section. Although the applicability condition is crucial for the soundness of the final rewriting, it may prevent the generation of queries that are vital for the completeness of the rewriting. This is illustrated in the following example.

Example 4.3 (*Incomplete Rewritings*). Consider the set Σ consisting of the TGDs

$$\sigma_1 : project(X), inArea(X, Y) \rightarrow \exists Z\, hasCollaborator(Z, Y, X),$$
$$\sigma_2 : hasCollaborator(X, Y, Z) \rightarrow collaborator(X),$$

and the CQ

$$q : p(B, C) \leftarrow \underbrace{hasCollaborator(A, B, C)}_{\underline{a}}, \underbrace{collaborator(A)}_{\underline{b}}.$$

The only viable strategy in this case is to apply σ_2 to $\{\underline{b}\}$, since σ_1 is not applicable to $\{\underline{a}\}$ due to the join variable A. The obtained query is

$$q' : p(B, C) \leftarrow hasCollaborator(A, B, C), hasCollaborator(A, E, F),$$

where E and F are fresh variables. Notice that the variable A remains a join variable and thus σ_1 is not applicable since the applicability condition is violated. However, q' has the same semantic meaning as

$$q'' : p(B, C) \leftarrow hasCollaborator(A, B, C),$$

in which A occurs only once. Since σ_1 is applicable to $body(q'')$ we get the query

$$q''' : p(B, C) \leftarrow project(C), inArea(C, B).$$

The query q'' is the result of unifying the body atoms of q', thus this unification step is critical for generating q'''. Let us now show that indeed q''' is crucial for the completeness of the final rewriting. Consider the database $D = \{project(a), inArea(a, b)\}$. Clearly, $chase(D, \Sigma) = D \cup \{hasCollaborator(z, b, a), collaborator(z)\}$, where $z \in \Gamma_N$, hence $\langle b, a\rangle \in ans(q, D, \Sigma)$. Observe that, without the query q''', there is no way to have the tuple $\langle b, a\rangle$ in the answer to the final rewriting over D, implying that q''' is needed for the completeness of the rewriting.

From the previous discussion we conclude that, apart from the rewriting step, an additional unification step is needed to convert some join variables into nonjoin ones. The purpose of this step, that we call the *factorization step*, is to satisfy the applicability condition and thereby guarantee the completeness of the final rewriting. To sum up, the perfect rewriting of a CQ q w.r.t. a set Σ of TGDs is computed by exhaustively applying the two steps discussed earlier, namely rewriting and factorization.

4.2. The Algorithm XRewrite

We proceed with the formal definition of our rewriting algorithm, called XRewrite. Before going into the details of the algorithm, we first need to formalize the applicability condition and the notion of factorizability. We assume without loss of generality that the variables occurring in queries and those appearing in TGDs constitute two disjoint sets. Given a CQ q, a variable is called *shared* in q if it occurs more than once in q. Notice that the distinguished variables of q are trivially shared since, by definition, they occur both in $body(q)$ and $head(q)$.

Definition 4.4 (*Applicability*). Consider a CQ q and a TGD σ. Given a set of atoms $S \subseteq body(q)$, we say that σ is *applicable* to S if the following conditions are satisfied:

(1) the set $S \cup \{head(\sigma)\}$ unifies, and
(2) for each $\underline{a} \in S$, if the term at position π in $\underline{a}$ is either a constant or a shared variable in q, then $\pi \neq \pi_{\exists}(\sigma)$.

Let us now focus on factorizability that will be the basis of the factorization step. Recall that the factorization step is necessary in order to convert some shared variables into nonshared ones, with the aim of satisfying the applicability condition. In general, this can be achieved by exhaustively unifying all the atoms that unify in the body of a query. However, some of these unifications do not contribute in any way to satisfying the applicability condition and, as a result, many superfluous queries are generated. We illustrate this situation by means of an example.

Example 4.5. Consider the following TGD and query:

$$\sigma : s(X) \rightarrow \exists Y\, r(X, Y) \qquad q : p(A) \leftarrow r(A, B), r(C, B), r(B, E).$$

Since σ is applicable to $\{r(B, E)\}$, we obtain the query

$$q' : p(A) \leftarrow \underbrace{r(A, B), r(C, B)}_{S}, s(B).$$

Due to the shared variable B, σ is not applicable to S. One can proceed with the unification of $r(A, B)$ and $r(C, B)$ in order to make B nonshared and satisfy the applicability condition; clearly, the query

$$q'' : p(A) \leftarrow r(A, B), s(B)$$

is obtained. However, the variable B is still shared and there is no way to make it nonshared. Thus, the unification of $r(A, B)$ and $r(C, B)$ does not contribute to satisfying the applicability condition and the query q'' is not needed.

Clearly, the exhaustive unification produces a nonnegligible number of redundant queries. It is thus necessary to apply a restricted form of factorization that generates a possibly small number of CQs that are vital for the completeness of the rewriting algorithm. This corresponds to the identification of all the atoms in the query whose shared existential variables come from the same atom in the chase, and they can be unified with no loss of information. Summing up, the key idea underlying our notion of factorizability is as follows: in order to apply the factorization step, there must exist a TGD that can be applied to its output.

Definition 4.6 (*Factorizability*). Consider a CQ q and a TGD σ. Given a set of atoms $S \subseteq body(q)$, where $|S| \geqslant 2$, we say that S is *factorizable* w.r.t. σ if the following conditions are satisfied:

(1) S unifies,
(2) $\pi_{\exists}(\sigma) \neq \varepsilon$, and

(3) there exists a variable $V \notin var(body(q) \setminus S)$ that occurs in every atom of S only at position $\pi_\exists(\sigma)$.

Example 4.7. Consider the TGD $\sigma : s(X), r(X, Y) \rightarrow \exists Z\, t(X, Y, Z)$ and the CQs

$$q_1 : p(A) \leftarrow \underbrace{t(a, A, C), t(B, a, C)}_{S_1},$$
$$q_2 : p(A) \leftarrow s(C), \underbrace{t(A, B, C), t(A, E, C)}_{S_2},$$
$$q_3 : p(A) \leftarrow \underbrace{t(A, B, C), t(A, C, C)}_{S_3},$$

where $a \in \Gamma$. The set S_1 is factorizable w.r.t. σ since the substitution $\{A \rightarrow a, B \rightarrow a\}$ is a unifier for S_1 and also since C appears in both atoms of S_1 only at position $\pi_\exists(\sigma) = t[3]$. On the other hand, S_2 and S_3, although they unify, are not factorizable w.r.t. σ since in q_2 the variable C occurs also outside S_2, while in q_3 the variable C appears not only at position $\pi_\exists(\sigma)$ but also at position $t[2]$.

Let us clarify that the notion of factorizability is incomparable to that of query minimization [Chandra and Merlin 1977]. Recall that the goal of query minimization is to construct a query that is equivalent to the original one and at the same time is minimal. Observe that q_1, given in Example 4.7, is already minimal since there is no endomorphism that can be applied on q_1 to make it smaller, but $S_1 \subseteq body(q_1)$ is factorizable w.r.t. σ and the obtained query is $p(A) \leftarrow t(a, a, C)$, which is not equivalent to q_1. On the other hand, q_2 is not minimal since, by applying the endomorphism $\{E \rightarrow B\}$, we get an equivalent and smaller query, but the factorization step is not applied.

Having the preceding key notions in place, we are now ready to present the algorithm XRewrite depicted in Algorithm 1. As said before, the perfect rewriting of a CQ q w.r.t. a set Σ of TGDs is computed by exhaustively applying (i.e., until a fixpoint is reached) the rewriting and the factorization steps. Notice that the CQs that are the result of the factorization step are nothing else than auxiliary queries critical for the completeness of the final rewriting, but are not needed in the final rewriting. Thus, during the iterative procedure, we label the queries with r (respectively, f) in order to keep track of which of them are generated by the rewriting (respectively, factorization) step. The input query, although not a result of the rewriting step, is labeled by r since it must be part of the final rewriting. Moreover, once we exhaustively apply on a CQ the two crucial steps, it is not necessary to revisit it, since this will lead to redundant queries. Hence, we also label the queries with e (respectively, u) indicating that a query is already explored (respectively, unexplored). Let us now describe the two main steps of the algorithm. In the sequel, consider a triple $\langle q, x, y \rangle$, where $\langle x, y \rangle \in \{\mathsf{r}, \mathsf{f}\} \times \{\mathsf{e}, \mathsf{u}\}$ (this is how we indicate that q is labeled by x and y) and a TGD $\sigma \in \Sigma$. We assume that q is of the form $p(\mathbf{X}) \leftarrow \varphi(\mathbf{X}, \mathbf{Y})$.

Rewriting Step. For each $S \subseteq body(q)$ such that σ is applicable to S, the i-th application of the rewriting step generates the query $q' = \gamma_{S,\sigma^i}(q[S/body(\sigma^i)])$, where σ^i is the TGD obtained from σ by replacing each variable X with X^i, γ_{S,σ^i} is the MGU for the set $S \cup \{head(\sigma^i)\}$ (which is the identity on the variables that appear in the body but not in the head of σ^i), and $q[S/body(\sigma^i)]$ is obtained from q by replacing S with $body(\sigma^i)$, namely is the query with $p(\mathbf{X})$ as its head and $(\varphi(\mathbf{X}, \mathbf{Y}) \setminus S) \cup body(\sigma^i)$ as its body. By considering σ^i (instead of σ) we actually rename, using the integer i, the variables of σ. This renaming step is needed in order to avoid undesirable clutter

ALGORITHM 1: The algorithm XRewrite

Input: a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$
Output: the perfect rewriting of q w.r.t. Σ

```
i := 0;
Q_REW := {⟨q, r, u⟩};
repeat
    Q_TEMP := Q_REW;
    foreach ⟨q, x, u⟩ ∈ Q_TEMP, where x ∈ {r, f} do
        foreach σ ∈ Σ do
            /* rewriting step                                              */
            foreach S ⊆ body(q) such that σ is applicable to S do
                i := i + 1;
                q' := γ_{S,σ^i}(q[S/body(σ^i)]);
                if there is no ⟨q'', r, ⋆⟩ ∈ Q_REW such that q' ≃ q'' then
                    Q_REW := Q_REW ∪ {⟨q', r, u⟩};
                end
            end
            /* factorization step                                          */
            foreach S ⊆ body(q) which is factorizable w.r.t. σ do
                q' := γ_S(q);
                if there is no ⟨q'', ⋆, ⋆⟩ ∈ Q_REW such that q' ≃ q'' then
                    Q_REW := Q_REW ∪ {⟨q', f, u⟩};
                end
            end
        end
        /* query q is now explored                                         */
        Q_REW := (Q_REW \ {⟨q, x, u⟩}) ∪ {⟨q, x, e⟩};
    end
until Q_TEMP = Q_REW;
Q_FIN := {q | ⟨q, r, e⟩ ∈ Q_REW};
return Q_FIN
```

among the variables introduced during different applications of the rewriting step. Finally, if there is no $\langle q'', \mathsf{r}, \star\rangle \in Q_{\text{REW}}$, that is, an (explored or unexplored) query which is a result of the rewriting step, such that q' and q'' are the same (modulo bijective variable renaming), denoted $q' \simeq q''$, then $\langle q', \mathsf{r}, \mathsf{u}\rangle$ is added to Q_{REW}.

Factorization Step. For each $S \subseteq body(q)$ that is factorizable w.r.t. σ, the factorization step generates the query $q' = \gamma_S(q)$, where γ_S is the MGU for S. If there is no $\langle q'', \star, \star\rangle \in Q_{\text{REW}}$, namely a query that is a result of the rewriting or the factorization step, and is explored or unexplored such that $q' \simeq q''$, then $\langle q', \mathsf{f}, \mathsf{u}\rangle$ is added to Q_{REW}.

It is important to say that, if the input set of TGDs is sticky, then both γ_{S,σ^i} and γ_S are defined in such a way that, for each of their mapping $V \to U$, $V \in var(q)$ implies $U \in var(q)$; their existence is guaranteed by stickiness (see the proof of Lemma 4.9). The reason why we employ these MGUs (instead of arbitrary ones) is to ensure a crucial syntactic property of each query generated during the rewriting process (see Lemma 4.9), that in turn will allow us to establish the termination of XRewrite under sticky sets of TGDs. Before we proceed further, let us briefly discuss the relationship between our approach and the one employed in König et al. [2012] that is based on the so-called piece unifier. Roughly, a piece-based rewriting step, the building block of the algorithm in König et al. [2012], simulates a factorization and a rewriting step of XRewrite. Let us illustrate this via a simple example.

Example 4.8. Consider the TGD and the CQ

$$\sigma : r(X) \rightarrow \exists Y\, s(X,Y) \qquad q : p \leftarrow \underbrace{s(A,B), s(C,B), s(C,D)}_{S}, t(A,C).$$

A pair (S, γ), where γ is an MGU for the set $S \cup \{head(\sigma)\}$, is called piece unifier of q with σ if: (i) the universally quantified variables of σ, denoted $var_\forall(\sigma)$, are mapped by γ to $var_\forall(\sigma)$; and (ii) each variable of $var(S) \cap var(body(q) \setminus S)$ is mapped by γ to $var_\forall(\sigma)$. Such an MGU is $\gamma = \{A \rightarrow X, B \rightarrow Y, C \rightarrow X, D \rightarrow Y\}$. The existence of the piece unifier (S, γ) implies that S can be rewritten at a single (piece-based) rewriting step using σ, thus the query $q' : p \leftarrow r(X), t(X,X)$ be obtained.

Now, observe that the set $\{s(A,B), s(C,B)\} \subseteq body(q)$ is factorizable w.r.t. σ and that after applying the factorization step we get the query $p \leftarrow s(A,B), s(C,D), t(A,C)$. Then, σ is applicable to $\{s(A,B), s(C,D)\}$ and, after applying the rewriting step, we get the query $p \leftarrow r(A), t(A,A)$ that coincides (modulo variable renaming) with q'.

4.3. Termination of XRewrite

Let us now establish the termination of XRewrite. We first establish a key syntactic property of the constructed rewritten query. In the sequel, for notational convenience, given a CQ q and a set Σ of TGDs, we denote by q_Σ the rewritten query XRewrite(q, Σ).

LEMMA 4.9. *Consider a CQ q over a schema $\mathcal{R}$, and a set Σ of TGDs over $\mathcal{R}$. For each $q' \in q_\Sigma$ the following hold.*

(1) If $\Sigma \in$ LINEAR, then $|body(q)| \geqslant |body(q')|$.
(2) If $\Sigma \in$ STICKY, then every variable of $(var(q') \setminus var(q))$ occurs only once in q'.

PROOF. Part (1) follows immediately by definition of linear TGDs. In particular, since each linear TGD has only one body atom, during the rewriting step we replace a set of atoms in the body of the CQ under consideration with a single atom. Notice that, during the factorization step, since we unify atoms, we always decrease the number of atoms in the body of the CQ.

Part (2) is established by induction on the number of applications of the rewriting and factorization steps. We denote by q^i_Σ the part of q_Σ obtained after i applications either of the factorization or the rewriting step. The proof is by induction on $i \geqslant 0$.

Base step. Clearly, $q^0_\Sigma = q$, and the claim holds trivially.

Inductive step. In case that $q^{i+1}_\Sigma = q^i_\Sigma$, where $i > 0$, the claim follows immediately by the induction hypothesis. The interesting case is when $q^{i+1}_\Sigma = q^i_\Sigma \cup \{p'\}$, where p' was obtained from a CQ $p \in q^i_\Sigma$ by applying either the rewriting or the factorization step. Henceforth, we refer to the variables (not occurring in q) introduced during the rewriting process as *new variables*. We identify the following two cases.

Case 1. First, assume that p' was obtained during the j-th application of the rewriting step, where $j \leqslant i+1$, because the TGD $\sigma \in \Sigma$ is applicable to a set $S \subseteq body(p)$. Since by the induction hypothesis each new variable in S occurs only once, we can assume without loss of generality that, for each mapping $V \rightarrow U$ of γ_{S,σ^j}, U is not a new variable introduced during the first $j-1$ applications of the rewriting step. Recall that, by construction, for each $V \rightarrow U$ of γ_{S,σ^j}, $V \in var(q)$ implies $U \in var(q)$. It is easy to see that such an MGU always exists. In particular, if γ_{S,σ^j} does not satisfy the previous condition, then we can redefine it as $\mu \circ \gamma_{S,\sigma^j}$, where μ is constructed as follows: for each $V \rightarrow U$ of γ_{S,σ^j}, if $V \in var(q)$, $U \notin var(q)$ and there is no mapping $U \rightarrow V'$ in μ, then we add to μ the mapping $U \rightarrow V$. We proceed by case analysis on the reason why a new variable may appear in p'. We identify the following two cases.

(1) A variable V occurs in $body(\sigma^j)$ but not in $head(\sigma^j)$. By construction, $V \to U \in \gamma_{S,\sigma^j}$ implies $U = V$. Thus V is a new variable that appears in $body(p')$. Since $\Sigma \in$ STICKY, V occurs in $body(\sigma^j)$ only once, hence V appears in p' only once.
(2) A new variable is $V \in var(S)$, $\gamma_{S,\sigma^j}(V) = U$, where U occurs in the body and in the head of σ^j and where there is no assertion $U \to V'$ in γ_{S,σ^j}, where $V' \in var(q)$. By the induction hypothesis, V occurs only once in p and thus does not occur in p'. Since U does not appear in the left-hand side of an assertion of γ_{S,σ^j}, we get that U is a new variable appearing in $body(p')$ due to the fact that it occurs in $body(\sigma^j)$ and $head(\sigma^j)$. Notice that U, after applying SMarking, is marked; thus U occurs only once in $body(\sigma^j)$ since $\Sigma \in$ STICKY. This implies that U appears in p' only once.

Case 2. Now, suppose that p' was obtained by applying the factorization step. This implies that there exists a set $S \subseteq body(p)$, where $|S| \geqslant 2$, that unifies, and $p' = \gamma_S(p)$. Recall that, by construction, for each mapping $V \to U$ of γ_S, $V \in var(q)$ implies $U \in var(q)$. The existence of such an MGU is guaranteed since, by the induction hypothesis, each new variable in S occurs only once. In fact, γ_S can be defined as the MGU for S', where S' is obtained as follows: if a new variable W occurs in an atom $\underline{a} \in S$ at position π and there exists a set $\{\underline{b}_1, \ldots, \underline{b}_n\}$, where $n \geqslant 1$ such that at position π of each $\underline{b}_i$ a variable $W_i \in var(q)$ occurs, then replace W with W_1. It is now straightforward to see, by definition of γ_S, that each new variable in p' occurs only once. □

We now show that our rewriting algorithm terminates under linear and sticky TGDs.

THEOREM 4.10. *Consider a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$. If $\Sigma \in$ LINEAR or $\Sigma \in$ STICKY, then* XRewrite(q, Σ) *terminates.*

PROOF. Assume first that $\Sigma \in$ LINEAR. By Lemma 4.9 we get that, for each $q' \in q_\Sigma$, $|body(q)| \geqslant |body(q')|$. This implies that each $q' \in q_\Sigma$ can be equivalently rewritten as a CQ with at most $k = |body(q)| \cdot arity(\mathcal{R})$ variables. Therefore q_Σ contains (modulo variable renaming) at most k variables. Since the maximum number of CQs that can be constructed using k variables and $|\mathcal{R}|$ predicates is finite and also since the algorithm does not drop queries that it has generated, the claim follows.

Suppose now that $\Sigma \in$ STICKY. Given a CQ $p \in q_\Sigma$, let $p^\star$ be the query obtained from p by replacing each variable of $var(p) \backslash var(q)$ with the symbol $\star$. Since by Lemma 4.9 each variable of $var(p) \backslash var(q)$ occurs only once in p, we get the following: for each pair of CQs p_1 and p_2 of q_Σ, if $p_1^\star = p_2^\star$, then p_1 and p_2 are the same modulo bijective variable renaming. Therefore, the maximum number of CQs that can be constructed during the execution of XRewrite is bounded by the number of different CQs that can be constructed using terms of $T = (terms(q) \cup \{\star\})$ and predicates of $\mathcal{R}$. Since both T and $\mathcal{R}$ are finite and also since the algorithm does not drop queries that it has generated, we conclude that XRewrite terminates under sticky sets of TGDs. □

Clearly, the check that the obtained query is not already present (modulo bijective variable renaming) each time the rewriting or the factorization step is applied is crucial in order to guarantee the termination of XRewrite. An alternative way, actually the one that we employ in the implementation of our algorithm, is to maintain an auxiliary set of CQs Q_{can} that stores the generated queries in a canonical form, that is, after applying a canonical renaming step, and to run the algorithm until a fixpoint of Q_{can} is reached. Formally, given a CQ q, assuming that Σ is the input set of TGDs and $\mathcal{R}$ the underlying schema, a canonical renaming $can_q : terms(body(q)) \to (\Gamma_q \cup \Delta_q)$, where $\Gamma_q \subset \Gamma$ are the constants occurring in q, and $\Delta_q \subset \Gamma_N$ is such that $(\Delta_q \cap var(q)) = \varnothing$, $|\Delta_q| = |body(q)| \cdot arity(\mathcal{R})$ if $\Sigma \in$ LINEAR, and $|\Delta_q| = |\mathcal{R}| \cdot (|terms(q)|+1)^{arity(\mathcal{R})} \cdot arity(\mathcal{R})$ if $\Sigma \in$ STICKY, is a one-to-one substitution mapping each constant of Γ_q to itself, and each

variable of $var(q)$ to the first unused element of Δ_q; a lexicographic order is assumed on Δ_q. It is easy to see that, given two CQs q and p, $can_q(q) = can_p(p)$ implies that q and p are the same query (modulo bijective variable renaming).

4.4. The Size of the Rewriting

By exploiting the analysis in the proof of Theorem 4.10, it is easy to establish an upper bound on the size of the rewriting constructed by XRewrite.

THEOREM 4.11. *Consider a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$. The following hold:*

(1) $|q_\Sigma| \in \mathcal{O}((|\mathcal{R}| \cdot (arity(\mathcal{R}) \cdot |body(q)|)^{arity(\mathcal{R})})^{|body(q)|})$ *if* $\Sigma \in$ LINEAR*, and*
(2) $|q_\Sigma| \in 2^{\mathcal{O}(|\mathcal{R}| \cdot (arity(\mathcal{R}) \cdot |body(q)|)^{arity(\mathcal{R})})}$ *if* $\Sigma \in$ STICKY.

PROOF. Assume first that $\Sigma \in$ LINEAR. As discussed in the proof of Theorem 4.10, the number of variables that can appear in q_Σ is bounded by $(arity(\mathcal{R}) \cdot |body(q)|)$. Thus the number of atoms that can appear in q_Σ is at most $|\mathcal{R}| \cdot (arity(\mathcal{R}) \cdot |body(q)|)^{arity(\mathcal{R})}$. Since $|body(q')| \leqslant |body(q)|$, for each $q' \in q_\Sigma$, we immediately get that $|q_\Sigma| \leqslant (|\mathcal{R}| \cdot (arity(\mathcal{R}) \cdot |body(q)|)^{arity(\mathcal{R})})^{|body(q)|}$; part (1) follows. Assume now that $\Sigma \in$ STICKY. As discussed in the proof of Theorem 4.10, the number of variables that can appear in q_Σ is bounded by $|terms(q)| + 1 \leqslant (arity(\mathcal{R}) \cdot |body(q)|) + 1$, hence the number of atoms that can appear in q_Σ is at most $|\mathcal{R}| \cdot ((arity(\mathcal{R}) \cdot |body(q)|) + 1)^{arity(\mathcal{R})}$. Since a CQ $q' \in q_\Sigma$ can have in its body any subset of these atoms, we conclude that $|q_\Sigma| \leqslant 2^{(|\mathcal{R}| \cdot ((arity(\mathcal{R}) \cdot |body(q)|)+1)^{arity(\mathcal{R})})}$; part (2) follows. □

An interesting question is whether the exponential (respectively, double-exponential) size of the UCQ rewriting is unavoidable when we consider linear (respectively, sticky) sets of TGDs. In what follows, we give an affirmative answer to this question.

THEOREM 4.12. *The following hold.*

(1) There exists a CQ q over a schema $\mathcal{R}$ and a set $\Sigma \in$ LINEAR over $\mathcal{R}$ such that, for any UCQ rewriting Q of q w.r.t. Σ, $|Q| \in .\Omega((|\mathcal{R}|)^{|body(q)|})$,
(2) There exists a CQ q over a schema $\mathcal{R}$ and a set $\Sigma \in$ STICKY over $\mathcal{R}$ such that, for any UCQ rewriting Q of q w.r.t. Σ, $|Q| \in \Omega(2^{(2^{arity(\mathcal{R})})})$.

PROOF. For part (1), let $\mathcal{R} = \{p_0, \ldots, p_m\}$ and consider the CQ and the set of TGDs

$$q \; : \; p \; \leftarrow \; p_0(A_1), \ldots, p_0(A_n) \qquad \Sigma \; = \; \big\{p_i(X) \; \rightarrow \; p_0(X)\big\}_{i \in [m]} .$$

It is not difficult to see that any UCQ rewriting of q w.r.t. Σ must contain a CQ q' such that $body(q') \in (\{p_i(A_1)\}_{i \in [m]} \times \{p_i(A_2)\}_{i \in [m]} \times \cdots \times \{p_i(A_n)\}_{i \in [m]})$. Since the cardinality of the aforesaid set is $m^n = (|\mathcal{R}|)^{|body(q)|}$, the claim follows.

For part (2), let $\mathcal{R} = \{p_0, \ldots, p_n, s, r\}$ and consider the atomic CQ $q : p \leftarrow p_0(0, \ldots, 0)$, where p_0 is an n-ary predicate, as well as the sticky set Σ of TGDs

$$\{p_i(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n), p_i(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_n) \rightarrow p_i(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n)\}_{i \in [n]},$$
$$\{s_i(X_1, \ldots, X_n) \; \rightarrow \; p_n(X_1, \ldots, X_n)\}_{i \in [2]}.$$

It is easy to verify that any UCQ rewriting of q w.r.t. Σ must contain a CQ q' such that $body(q') \in \times_{\mathbf{t} \in \{0,1\}^n} \{s_1(\mathbf{t}), s_2(\mathbf{t})\}$, and $|\times_{\mathbf{t} \in \{0,1\}^n} \{s_1(\mathbf{t}), s_2(\mathbf{t})\}| = 2^{(2^n)} = 2^{(2^{arity(\mathcal{R})})}$. □

4.5. Correctness of XRewrite

We now establish the correctness of XRewrite. Towards this aim, two auxiliary technical lemmas are needed. The first one, which is used for soundness, states that the answer to the final rewriting is a subset of the answer to the input query. In what follows, let $\mathbf{X}^i$ be the sequence of variables obtained by replacing each variable X of $\mathbf{X}$ with X^i.

LEMMA 4.13. *Consider a CQ q over a schema $\mathcal{R}$, a database D for $\mathcal{R}$, and a set Σ of TGDs over $\mathcal{R}$. It holds that $ans(q_\Sigma, D, \Sigma) \subseteq ans(q, D, \Sigma)$.*

PROOF. It suffices to show that, for a tuple of constants $\mathbf{t}$, $\mathbf{t} \in ans(q_\Sigma, D, \Sigma)$ implies $\mathbf{t} \in ans(q, D, \Sigma)$, or, equivalently, $\mathbf{t} \in q_\Sigma(chase(D, \Sigma))$ implies $\mathbf{t} \in q(chase(D, \Sigma))$. It is straightforward to see that the factorization step does not affect the soundness of our algorithm. Thus, we assume without loss of generality that q_Σ is the UCQ constructed without applying the factorization step. We denote by q_Σ^i the part of q_Σ obtained after $i \geqslant 0$ applications of the rewriting step. The proof is by induction on i.

Base step. Clearly, $q_\Sigma^0 = q$, and the claim holds trivially.

Inductive step. Suppose now that $\mathbf{t} \in q_\Sigma^i(chase(D, \Sigma))$, for $i \geqslant 0$. This implies that there exists $p \in q_\Sigma^i$ and a homomorphism h such that $h(body(p)) \subseteq chase(D, \Sigma)$ and $h(\mathbf{V}) = \mathbf{t}$, where $\mathbf{V}$ are the distinguished variables of p. If $p \in q_\Sigma^{i-1}$, then the claim follows by the induction hypothesis. The interesting case is when p was obtained during the i-th application of the rewriting step from a CQ $p' \in q_\Sigma^{i-1}$, namely $q_\Sigma^i = q_\Sigma^{i-1} \cup \{p\}$. By the induction hypothesis, it suffices to show that $\mathbf{t} \in q_\Sigma^{i-1}(chase(D, \Sigma))$. Clearly, there exists a TGD $\sigma \in \Sigma$ of the form $\varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists Z r(\mathbf{X}, Z)$ that is applicable to a set $S \subseteq body(p')$ and p is the query $\gamma(p'[S/body(\sigma^i)])$; let γ be the MGU for $S \cup \{head(\sigma^i)\}$. Observe that $h(\gamma(\varphi(\mathbf{X}^i, \mathbf{Y}^i))) \subseteq chase(D, \Sigma)$, hence σ is applicable to $chase(D, \Sigma)$; let $\mu = h \circ \gamma$. Thus $\mu'(r(\mathbf{X}^i, Z^i)) \in chase(D, \Sigma)$, where $\mu' \supseteq \mu|_{\mathbf{X}^i}$. We define the substitution $h' = h \cup \{\gamma(Z^i) \rightarrow \mu'(Z^i)\}$. To establish that h' is well defined, it suffices to show that $\gamma(Z^i) \notin \Gamma$ and also that there is no mapping $V \rightarrow U \in h$ such that $\gamma(Z^i) = V$. Towards a contradiction, suppose that $\gamma(Z^i)$ is either a constant or appears in the left-hand side of an assertion of h. It is easy to verify that in this case there exists an atom $\underline{a} \in S$ such that at position $\pi_\exists(\sigma)$ in $\underline{a}$ occurs either a constant or a variable that is shared in p'. But this contradicts the fact that σ is applicable to S, hence h' is well defined. It remains to show that the substitution $h' \circ \gamma$ maps $body(p')$ to $chase(D, \Sigma)$ and $h'(\gamma(\mathbf{V}')) = \mathbf{t}$, where $\mathbf{V}'$ are the distinguished variables of p'; this immediately implies that $\mathbf{t} \in q_\Sigma^{i-1}(chase(D, \Sigma))$. Clearly, $\gamma(body(p') \setminus S) \subseteq body(p)$. Since $h(body(p)) \subseteq chase(D, \Sigma)$, we get that $h'(\gamma(body(p') \setminus S)) \subseteq chase(D, \Sigma)$. Moreover, $h'(\gamma(S)) = h'(\gamma(r(\mathbf{X}^i, Z^i))) = r(h'(\gamma(\mathbf{X}^i)), h'(\gamma(Z^i))) = r(\mu(\mathbf{X}^i), \mu'(Z^i)) = \mu'(r(\mathbf{X}^i, Z^i)) \in chase(D, \Sigma)$. Finally, since $\gamma(\mathbf{V}') = \mathbf{V}$ and $h(\mathbf{V}) = \mathbf{t}$, we get that $h'(\gamma(\mathbf{V}')) = \mathbf{t}$. □

The second auxiliary lemma asserts that the answer to the final rewriting is a subset of the set of tuples obtained by simply evaluating it over the input database.

LEMMA 4.14. *Consider a CQ q over a schema $\mathcal{R}$, a database D for $\mathcal{R}$, and a set Σ of TGDs over $\mathcal{R}$. It holds that $ans(q_\Sigma, D, \Sigma) \subseteq q_\Sigma(D)$.*

PROOF. It suffices to show that, for a tuple of constants $\mathbf{t}$, $\mathbf{t} \in ans(q_\Sigma, D, \Sigma)$ implies $\mathbf{t} \in q(D)$, or, equivalently, $\mathbf{t} \in q_\Sigma(chase(D, \Sigma))$ implies $\mathbf{t} \in q_\Sigma(D)$. We proceed by induction on the number of applications of the chase step.

Base step. Clearly, $chase^{[0]}(D, \Sigma) = D$ and the claim trivially holds.

Inductive step. Suppose now that $\mathbf{t} \in q_\Sigma(chase^{[i]}(D, \Sigma))$, for $i \geqslant 0$. This implies that there exists $p \in q_\Sigma$ and a homomorphism h such that $h(body(p)) \subseteq chase^{[i]}(D, \Sigma)$ and $h(\mathbf{V}) = \mathbf{t}$, where $\mathbf{V}$ are the distinguished variables of p. If $h(body(p)) \subseteq chase^{[i-1]}(D, \Sigma)$,

then the claim follows by the induction hypothesis. The nontrivial case is when the atom $\underline{a}$, obtained during the i-th application of the chase step by applying a TGD $\sigma : \varphi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists Z r(\mathbf{X}, Z)$, belongs to $h(body(p))$. Clearly, there exists a homomorphism μ such that $\mu(\varphi(\mathbf{X}, \mathbf{Y})) \subseteq chase^{[i-1]}(D, \Sigma)$ and $\underline{a} = \mu'(r(\mathbf{X}, \mathbf{Y}))$, where $\mu' \supseteq \mu|_{\mathbf{X}}$. By the induction hypothesis, it suffices to show that $\mathbf{t} \in q_\Sigma(chase^{[i-1]}(D, \Sigma))$. Before we proceed further, we need an auxiliary claim. Its proof can be found in electronic Appendix B.2.

CLAIM 4.15. *There exists a CQ $p' \in q_\Sigma$ and a set of atoms $S \subseteq body(p')$ such that σ is applicable to S and also there exists a homomorphism λ such that $\lambda(body(p')\backslash S) \subseteq chase^{[i-1]}(D, \Sigma)$, $\lambda(\mathbf{V}') = \mathbf{t}$, where $\mathbf{V}'$ are the distinguished variables of p', and $\lambda(S) = \underline{a}$.*

The previous claim implies that there exists $i \geqslant 1$ such that during the rewriting process eventually we will get a CQ p'' with $body(p'') = \gamma(body(p')\backslash S) \cup \gamma(\varphi(\mathbf{X}^i, \mathbf{Y}^i))$, where γ is the MGU for $S \cup \{head(\sigma^i)\}$. It remains to show that there exists a homomorphism that maps $body(p'')$ to $chase^{[i-1]}(D, \Sigma)$ and the distinguished variables $\mathbf{V}''$ of p'' to $\mathbf{t}$. Since $\lambda \cup \mu'$ is a well-defined substitution, it is a unifier for $S \cup \{head(\sigma^i)\}$. By definition of the MGU, there exists a substitution θ such that $\lambda \cup \mu' = \theta \circ \gamma$. Observe that $\theta(body(p'')) = \theta(\gamma(body(p')\backslash S) \cup \gamma(\varphi(\mathbf{X}^i, \mathbf{Y}^i))) = (\lambda \cup \mu')(body(p')\backslash S) \cup (\lambda \cup \mu')(\varphi(\mathbf{X}^i, \mathbf{Y}^i)) = \lambda(body(p')\backslash S) \cup \mu'(\varphi(\mathbf{X}^i, \mathbf{Y}^i)) \subseteq chase^{[i-1]}(D, \Sigma)$. Finally, $\theta(\mathbf{V}'') = \theta(\gamma(\mathbf{V}')) = (\lambda \cup \mu')(\mathbf{V}') = \lambda(\mathbf{V}') = \mathbf{t}$. □

We are now ready to establish the soundness and completeness of XRewrite.

THEOREM 4.16. *Consider a CQ q over a schema $\mathcal{R}$, a database D for $\mathcal{R}$, and a set Σ of TGDs over $\mathcal{R}$. It holds that $q_\Sigma(D) = ans(q, D, \Sigma)$.*

PROOF. Since $D \subseteq chase(D, \Sigma)$, by monotonicity of CQs, $q_\Sigma(D) \subseteq q_\Sigma(chase(D, \Sigma))$, in turn implying that $q_\Sigma(D) \subseteq ans(q_\Sigma, D, \Sigma)$. By Lemma 4.13, we immediately get that $q_\Sigma(D) \subseteq ans(q, D, \Sigma)$. Conversely, since $q \in q_\Sigma$, we get that $ans(q, D, \Sigma) \subseteq ans(q_\Sigma, D, \Sigma)$. Lemma 4.14 implies that $ans(q, D, \Sigma) \subseteq q_\Sigma(D)$ and the claim follows. □

Let us conclude this section by noticing that XRewrite can treat even more expressive classes of TGDs than linear and sticky TGDs, namely multilinear [Calì et al. 2012a] and sticky-join [Calì et al. 2012b] TGDs that guarantee the first-order rewritability of CQ answering. The goal of multilinearity was the definition of a natural formalism that is strictly more expressive than DL-Lite$_{\mathcal{R},\sqcap}$, that is, the extended version of DL-Lite$_{\mathcal{R}}$ allowing for concept conjunction [Calvanese et al. 2013b]. Sticky-joiness is the result of combining linearity and stickiness, with the aim of identifying more expressive classes of TGDs. For more details, see electronic Appendix B.3.

5. PARALLELIZE THE REWRITING PROCEDURE

An interesting question that comes up is whether the overall time that we need to compute the final rewriting can be reduced by designing a parallel version of XRewrite that exploits multicore architectures. In this section, we present some preliminary ideas and results regarding the parallelization of our algorithm. To the best of our knowledge, this is the first attempt to design a parallel rewriting algorithm. The key idea is to decompose the query q into smaller queries $q_1, \ldots, q_m$, where $m \geqslant 1$, in such a way that, if a variable V occurs in at least two queries of $\{q_1, \ldots, q_m\}$, then each occurrence of V occurs at a position that may host only constants (in the instance constructed by the chase procedure). This allows us to independently rewrite each query q_i into Q_{q_i}, where $i \in [m]$, and then to merge the queries $Q_{q_1}, \ldots, Q_{q_m}$ in order to obtain the final rewriting. Notice that the decomposition technique described before is a new form of query decomposition that, in contrast to traditional methods such as the ones in Chekuri and Rajaraman [2000] and Gottlob et al. [2002], takes into account a

given set of TGDs and is engineered to be used for parallelizing our rewriting algorithm. Instead, the aim of existing techniques is to suggest an efficient strategy for executing the given query. Let us first give an informal description of our parallel procedure.

5.1. An Informal Description

Consider the following relational schema representing financial information about companies and their stocks.

stock(id, name, unit_price)	*company*(name, country, segment)
listComponent(stock, list)	*stockPortfolio*(company, stock, quantity)
finIndex(name, type, reference_market)	*hasStock*(stock, comany)
finInstrument(stock)	*legalPerson*(company).

Let Σ be the set consisting of the following linear TGDs; for clarity, we use more than one existentially quantified variable in the rule heads.

$$\begin{aligned}
\sigma_1 &: \mathit{stockPortfolio}(X, Y, Z) \rightarrow \exists V \exists W\, \mathit{company}(X, V, W)\\
\sigma_2 &: \mathit{stockPortfolio}(X, Y, Z) \rightarrow \exists V \exists W\, \mathit{stock}(Y, V, W)\\
\sigma_3 &: \mathit{listComponent}(X, Y) \rightarrow \exists Z \exists W\, \mathit{finIndex}(Y, Z, W)\\
\sigma_4 &: \mathit{listComponent}(X, Y) \rightarrow \exists Z \exists W\, \mathit{stock}(X, Z, W)\\
\sigma_5 &: \mathit{stockPortfolio}(X, Y, Z) \rightarrow \mathit{hasStock}(Y, X)\\
\sigma_6 &: \mathit{hasStock}(X, Y) \rightarrow \exists Z\, \mathit{stockPortfolio}(Y, X, Z)\\
\sigma_7 &: \mathit{stock}(X, Y, Z) \rightarrow \exists V \exists W\, \mathit{stockPortfolio}(V, X, W)\\
\sigma_8 &: \mathit{stock}(X, Y, Z) \rightarrow \mathit{finInstrument}(X)\\
\sigma_9 &: \mathit{company}(X, Y, Z) \rightarrow \mathit{legalPerson}(X).
\end{aligned}$$

The TGDs σ_1, σ_2, σ_3, and σ_4 set the "domain" and the "range" of the *stockPortfolio* and *listComponent* relations, respectively. The TGDs σ_5 and σ_6 assert that *stockPortfolio* and *hasStock* are "inverse relations", while σ_7 expresses that each stock must belong to a stock portfolio. The TGDs σ_8 and σ_9 model taxonomic relationships, in particular that each stock is a financial instrument and each company is a legal person. Consider also the following conjunctive query q asking for all the triples $\langle a, b, c\rangle$, where a is a financial instrument owned by the company b and listed on c:

$$\begin{aligned}
p(A, B, C) \leftarrow\ & \mathit{finInstrument}(A), \mathit{stockPortfolio}(B, A, D), \mathit{company}(B, E, F),\\
& \mathit{listComponent}(A, C), \mathit{finIndex}(C, G, H).
\end{aligned}$$

Recall that our intention is to decompose q into smaller subqueries in such a way that, if a variable V occurs in at least two such subqueries, then each occurrence of V occurs at a position that may host only constants (in the instance constructed by the chase procedure). After a careful inspection of the set Σ, it is easy to verify that, for every database D, if q is mapped to $\mathit{chase}(D, \Sigma)$ via a homomorphism h, then the only join variable occurring in q that can be mapped by h to a null value is B. More precisely, due to σ_7 a null value may appear at position *stockPortfolio*[1], which in turn may be propagated to position *company*[1] after applying σ_1. These positions are called affected w.r.t. σ_7, intuitively meaning that they can have a null generated by σ_7. The fact that only B appears at an affected position allows us to decompose q into four subqueries and then independently rewrite each one of them. The result of such a decomposition,

called existential-join decomposition, is the following:

$$q_1 : p_1(A) \leftarrow \mathit{finInstrument}(A)$$
$$q_2 : p_2(A, B) \leftarrow \mathit{stockPortfolio}(B, A, D), \mathit{company}(B, E, F)$$
$$q_3 : p_3(A, C) \leftarrow \mathit{listComponent}(A, C)$$
$$q_4 : p_4(C) \leftarrow \mathit{finIndex}(C, G, H).$$

Notice that, for each subquery q_i, the distinguished variables of q_i are the shared variables of q that appear outside $body(q_i)$, namely in $head(q)$ or in $body(q) \setminus body(q_i)$. The rewriting of q_i w.r.t. Σ, for each $i \in [4]$, is denoted Q_{q_i}. The last step is to merge the queries $Q_{q_1}, \ldots, Q_{q_4}$. This can be done via the reconciliation rule

$$\rho : p(A, B, C) \leftarrow p_1(A), p_2(A, B), p_3(A, C), p_4(C),$$

that intuitively says that the rewriting of q w.r.t. Σ is obtained by computing the cartesian product of the queries $Q_{q_1}, \ldots, Q_{q_4}$, while the variables A and C, which occur in more than one component, have the same semantic meaning, namely that the joins among different components are preserved. More precisely, the final rewriting of q w.r.t. Σ is obtained by unfolding the nonrecursive Datalog query $\langle Q_{q_1} \cup \cdots \cup Q_{q_4} \cup \{\rho\}, p\rangle$.

The UCQ obtained by employing the aforesaid technique and XRewrite(q, Σ) have exactly the same size. In other words, the parallelization of the rewriting procedure does not affect the size of the final rewriting. However, it significantly affects the execution time of the rewriting algorithm. The execution of XRewrite on q and Σ takes 194ms, whereas the execution of the parallel version of XRewrite takes 81ms (47ms for constructing $\langle Q_{q_1} \cup \ldots \cup Q_{q_4} \cup \{\rho\}, p\rangle$ and 34ms for unfolding it).

5.2. The Algorithm XRewriteParallel

Let us now formalize the idea discussed earlier. First, we need to define the notion of affected positions.

Definition 5.1 (*Affected Positions*). Consider a set Σ of TGDs over a schema $\mathcal{R}$. An *affected position* of $\mathcal{R}$ w.r.t. a pair $\langle \sigma, \Sigma\rangle$, where $\sigma \in \Sigma$, is defined inductively as follows:

(1) the position $\pi_\exists(\sigma)$ is affected w.r.t. $\langle \sigma, \Sigma\rangle$, and
(2) a position π in the head of a TGD $\sigma' \in \Sigma$ is affected w.r.t. $\langle \sigma, \Sigma\rangle$ if the same variable appears at π and in the $body(\sigma')$ only at positions affected w.r.t. $\langle \sigma, \Sigma\rangle$.

Example 5.2. Consider the set Σ of TGDs consisting of

$$\sigma_1 : p(X, Y), s(Y, Z) \rightarrow \exists W\, t(Y, X, W) \qquad \sigma_2 : t(X, Y, Z) \rightarrow \exists W\, p(W, Z).$$

It is easy to verify that

$$\langle \sigma_1, \Sigma\rangle = \{t[3], p[2]\} \qquad \langle \sigma_2, \Sigma\rangle = \{p[1], t[2]\}.$$

Notice that, although the variable Y in $body(\sigma_1)$ occurs at position $p[2] \in \langle \sigma_1, \Sigma\rangle$, $t[1]$ is not affected w.r.t. $\langle \sigma_1, \Sigma\rangle$ since Y also occurs at position $s[1] \notin \langle \sigma_1, \Sigma\rangle$.

By having the previous auxiliary notion in place, we are now ready to define the key notion of the existential-join decomposition of a CQ w.r.t. a set of TGDs.

Definition 5.3 (*Existential-Join Decomposition*). Consider a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$. An *existential-join decomposition* of q w.r.t. Σ is a partition P of $body(q)$ such that the following holds: if a variable $V \in var(q)$ occurs in $body(q)$ only at positions affected w.r.t. $\langle \sigma, \Sigma\rangle$ for some $\sigma \in \Sigma$, then there exists $S \in P$ such that $V \in var(S)$ and $V \notin var(P \setminus S)$. We say that P is an *optimal*, if there is no $S \in P$ such that $(P \setminus S) \cup \{S_1, S_2\}$, where $\{S_1, S_2\}$ is a partition of S, existential-join decomposition of q w.r.t. Σ.

ALGORITHM 2: The algorithm XRewriteParallel

Input: a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$
Output: the perfect rewriting of q w.r.t. Σ

```
/* decomposition step                                              */
⟨{q_1, ..., q_m}, ρ⟩ := decompose(q, Σ);
/* parallel step                                                   */
for q ∈ {q_1, ..., q_m} do in parallel
  |  Q_q := XRewrite(q, Σ);
end
/* merging step                                                    */
Π := Q_{q_1} ∪ ... ∪ Q_{q_m} ∪ {ρ};
Q_FIN := unfold(⟨Π, p⟩);
return Q_FIN
```

It can be proven that the optimal existential-join decomposition of a CQ w.r.t. a set of TGDs is unique. We are now ready to describe the parallel version of XRewrite. As already said, the key idea hinges on the fact that each component of an existential-join decomposition can be rewritten independently and the final rewriting obtained by merging the obtained rewritings via a reconciliation (Datalog) rule. Consider a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$; for notational convenience, we assume that $p(\mathbf{X})$ is the head atom of q and $var(q) = \{V_1, \ldots, V_n\}$. The parallel version of XRewrite, called XRewriteParallel and depicted in Algorithm 2, consists of the following three steps.

Decomposition Step. The optimal existential-join decomposition P of q w.r.t. Σ is computed; let $P = \{C_1, \ldots, C_m\}$. Then, for each $i \in [m]$ we construct the CQ

$$q_i \ : \ p_i(f_i(V_1, \ldots, V_n)) \ \leftarrow \ C_i,$$

where p_i is an auxiliary predicate not occurring in $\mathcal{R}$ and where $f_i(V_1, \ldots, V_n)$ is defined as the tuple $\langle V_{j_1}, \ldots, V_{j_k} \rangle$, where $1 \leqslant k \leqslant n$, such that: *(i)* $1 \leqslant j_1 < \cdots < j_k \leqslant n$, and *(ii)* for each $\ell \in [k]$, $V_{j_\ell} \in var(C_i) \cap (\mathbf{X} \cup (var(q) \setminus var(C_i)))$. Intuitively, $f_i(V_1, \ldots, V_n)$ is obtained from $\langle V_1, \ldots, V_n \rangle$ by keeping only those variables of $var(C_i)$ that are also distinguished variables of q, or that occur in a component other than C_i. Moreover, the reconciliation (Datalog) rule

$$\rho \ : \ p(\mathbf{X}) \ \leftarrow \ p_1(f_1(V_1, \ldots, V_n)), \ldots, p_m(f_m(V_1, \ldots, V_n))$$

is constructed. The decomposition step is carried out by the decompose function that accepts as input the query q and the set of TGDs Σ, then returns as output the pair $\langle \{q_1, \ldots, q_m\}, \rho \rangle$.

Parallel Step. We construct in m parallel computations the perfect rewriting Q_q of each CQ $q \in \{q_1, \ldots, q_m\}$ w.r.t. Σ by exploiting the rewriting algorithm XRewrite.

Merging Step. It is not difficult to verify that $\langle \Pi, p \rangle$, where $\Pi = (Q_{q_1} \cup \ldots \cup Q_{q_m} \cup \{\rho\})$, is a nonrecursive Datalog query. It is well known that such a query can be *unfolded* into a (finite) UCQ; for more details, see, for example, Abiteboul et al. [1995]. The perfect rewriting of the input CQ q w.r.t. Σ is the UCQ obtained by unfolding $\langle \Pi, p \rangle$, which is carried out by the unfold function.

It is easy to see that XRewriteParallel terminates under linear and sticky sets of TGDs. The decomposition step terminates since q and Σ are finite, the parallel step

since XRewrite terminates under linear and sticky sets of TGDs, and the merging step since the unfolding of a finite nonrecursive Datalog query is finite.

THEOREM 5.4. *Consider a CQ q over a schema $\mathcal{R}$ and a set Σ of TGDs over $\mathcal{R}$. If $\Sigma \in$ LINEAR or $\Sigma \in$ STICKY, then* XRewriteParallel(q, Σ) *terminates.*

The soundness and completeness of XRewriteParallel follows by construction. Instead of giving a formal proof (which is rather long and uninteresting), we intuitively explain why XRewriteParallel is sound and complete. For brevity, given a CQ q and a set Σ of TGDs, we denote by $q^{\shuffle}_{\Sigma}$ the rewritten query XRewriteParallel(q, Σ). It is possible to show that $q^{\shuffle}_{\Sigma}$ and q_{Σ} are the same (modulo bijective variable renaming), which immediately implies the soundness and completeness of XRewriteParallel. Let $P = \{C_1, \ldots, C_m\}$ be the optimal existential-join decomposition of q w.r.t. Σ. Each rewriting step applied during the execution of XRewriteParallel(q, Σ) corresponds to a rewriting step of XRewrite(q, Σ). This holds since, by construction of each $q_i : p_i(f_i(V_1, \ldots, V_n)) \leftarrow C_i$ where $V_1, \ldots, V_n$ are the variables of $var(q)$, a variable $V \in var(C_i)$ that is shared in q is also shared in q_i. More precisely, if V is a distinguished variable of q or occurs in a component of P other than C_i, then it also occurs in $head(q_i)$ and thus is shared in q_i; otherwise, if it occurs only in C_i, then it is trivially shared in q_i since, by hypothesis, it occurs more than once in C_i. Conversely, each rewriting step applied during the execution of XRewrite(q, Σ) corresponds to a rewriting step of XRewriteParallel(q, Σ). Towards a contradiction, assume that the preceding claim does not hold. This implies that, during the execution of XRewriteParallel(q, Σ), a valid rewriting step is not applied due to a missing factorization step. But this implies that a variable occuring in $body(q)$ only at those positions affected w.r.t. $\langle \sigma, \Sigma \rangle$, for some $\sigma \in \Sigma$, appears in more than one component of P—a contradiction. Notice that the reconciliation rule preserves the joins among different components of P and the claim follows.

THEOREM 5.5. *Consider a CQ q over a schema $\mathcal{R}$, a database for $\mathcal{R}$, and a set Σ of TGDs over $\mathcal{R}$. It holds that $q^{\shuffle}_{\Sigma}(D) = ans(q, D, \Sigma)$.*

6. OPTIMIZE THE REWRITING FOR LINEAR TGDS

Linearity of TGDs allows us to effectively identify atoms in the body of a query that are logically implied (w.r.t. a given set of TGDs) by other atoms in the same query. By exploiting this fact, we propose a technique, called *query elimination*, aiming at optimizing the obtained rewritten query under the class of linear TGDs. As we shall see in the experimental section, query elimination (which is an additional step during the execution of XRewrite) reduces: *(i)* the number of CQs of the perfect rewriting, *(ii)* the number of atoms in each query of the rewriting, and *(iii)* the number of joins to be executed. Let us first give a motivating example that exposes the key idea underlying query elimination and also illustrates its impact on the final rewriting.

6.1. A Motivating Example

Consider the set Σ of linear TGDs and the CQ q given in Section 5.1. The complete rewriting of q w.r.t. Σ contains 60 conjunctive queries executing 300 joins. However, by exploiting the set of TGDs, it is possible to eliminate redundant atoms in the generated queries and thus reduce the size of the final rewriting. For example, it is possible to eliminate from the given query q the atom $finInstrument(A)$ since, due to the existence of the TGDs σ_2 and σ_8 in Σ, if the atom $stockPortfolio(B, A, D)$ is satisfied, then immediately the atom $finInstrument(A)$ is also satisfied. Notice that, by eliminating a redundant atom from a query, we also eliminate all those queries generated starting from it during the rewriting process. Moreover, due to the TGD σ_3, if the atom $listComponent(A, C)$ in q is satisfied, then the atom $finIndex(C, G, H)$ is also satisfied and therefore can be

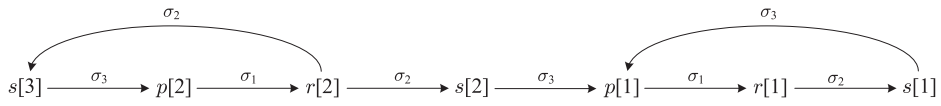

Fig. 4. Propagation graph for Example 6.2.

eliminated. Finally, due to the TGD σ_1, if the atom $stockPortfolio(B, A, D)$ is satisfied, then the atom $company(B, E, F)$ is also satisfied, hence the latter is redundant. The query that has to be considered as input of the rewriting process is therefore

$$p(A, B, C) \leftarrow stockPortfolio(B, A, D), listComponent(A, C),$$

thus producing a perfect rewriting containing the following two conjunctive queries executing only two joins:

$$p(A, B, C) \leftarrow listComponent(A, C), stockPortfolio(B, A, D)$$
$$p(A, B, C) \leftarrow listComponent(A, C), hasStock(A, B).$$

It is evident that, by eliminating redundant atoms from a query as described previously, we reduce the number of CQs of the perfect rewriting, the number of atoms in each query of the rewriting, and the number of joins to be executed.

6.2. Atom Coverage

Before formalizing the idea described before, let us first introduce some auxiliary technical notions.

Definition 6.1 (*Propagation Graph*). Consider a set Σ of TGDs over a schema $\mathcal{R}$. The *propagation graph* of Σ, denoted $PG(\Sigma)$, is a labeled directed multigraph $\langle N, E, \lambda \rangle$, where N is the node set, E is the edge set, and λ is a labeling function $E \rightarrow \Sigma$. The node set is the set of positions of $\mathcal{R}$. If there exists $\sigma \in \Sigma$ such that the same variable appears at position π_b in $body(\sigma)$ and at position π_h in $head(\sigma)$, then the edge $e = \langle \pi_b, \pi_h \rangle$ belongs to E with $\lambda(e) = \sigma$; no other edges belong to E.

The propagation graph of a set of linear TGDs encodes all the possible ways of propagating a term from one position to another during the chase. More precisely, the existence of a path from π_1 to π_2 implies that there may be a way to propagate a term from π_1 to π_2 during the construction of the chase. Given a path $P = v_1 \dots v_n$ where $n > 1$, of $PG(\Sigma) = \langle N, E, \lambda \rangle$, we say that P is *minimal* if the following condition is satisfied: there is no $1 < i < n$ and $0 < j < i$ such that $v_{i-j} \dots v_i = v_i \dots v_{i+j}$ and $\lambda(\langle v_{i-j}, v_{i-j+1} \rangle) \dots \lambda(\langle v_{i-1}, v_i \rangle) = \lambda(\langle v_i, v_{i+1} \rangle) \dots \lambda(\langle v_{i+j-1}, v_{i+j} \rangle)$. The minimality condition guarantees that cycles occurring in $PG(\Sigma)$ are traversed at most once.

Example 6.2. Consider the set Σ of linear TGDs consisting of

$$\sigma_1 : p(X, Y) \rightarrow \exists Z r(X, Y, Z) \quad \sigma_2 : r(X, Y, c) \rightarrow s(X, Y, Y) \quad \sigma_3 : s(X, X, Y) \rightarrow p(X, Y).$$

The propagation graph of Σ (without the isolated node $r[3]$) is depicted in Figure 4. The path $P = v_1 \dots v_6$, where $v_1 v_2 v_3 = v_4 v_5 v_6 = s[3]p[2]r[2]$ is minimal. However, the path $P' = v_1 \dots v_9$, where $v_1 v_2 v_3 = v_4 v_5 v_6 = v_7 v_8 v_9 = s[3]p[2]r[2]$ is not minimal, since the minimality condition is violated with $i = 4$ and $j = 3$; clearly, $v_1 v_2 v_3 v_4 = v_4 v_5 v_6 v_7 = s[3]p[2]r[2]s[3]$ and $\lambda(\langle v_1, v_2 \rangle)\lambda(\langle v_2, v_3 \rangle)\lambda(\langle v_3, v_4 \rangle) = \lambda(\langle v_4, v_5 \rangle)\lambda(\langle v_5, v_6 \rangle)\lambda(\langle v_6, v_7 \rangle) = \sigma_3\sigma_2\sigma_1$, which intuitively means that the cycle $s[3]p[2]r[2]s[3]$ occurs in P' twice.

Unfortunately, the existence of a path P from π_1 to π_2 does not guarantee the propagation of a term from π_1 to π_2. For example, consider the TGDs $\sigma_1 : r(X, Y) \rightarrow \exists Z t(Y, Z)$ and $\sigma_2 : t(X, X) \rightarrow s(X)$. It is easy to verify that, although in $PG(\{\sigma_1, \sigma_2\})$ the path

$r[2]t[1]s[1]$ exists, there is no way to propagate a term from $r[2]$ to $s[1]$ since the atom obtained by applying σ_1 does not trigger σ_2. Thus, the existence of such a path P guarantees the propagation of a term from π_1 to π_2 providing that, for each pair of consecutive edges $e = \langle \pi, \pi' \rangle$ and $e' = \langle \pi', \pi'' \rangle$ of P, where e and e' are labeled by the TGDs σ and σ', respectively, the atom obtained during the chase by applying σ triggers σ'. It is easy to verify that a natural sufficient condition for the latter is as follows: for each pair of consecutive edges e and e' of P that are labeled by σ and σ', respectively, there exists a homomorphism h such that $h(body(\sigma')) \subseteq head(\sigma)$; notice that this condition heavily relies on the linearity of the TGDs. A sequence $\sigma_1, \ldots, \sigma_n$ of linear TGDs where $n > 1$ is called *tight* if, for each $i \in [n-1]$, there exists a homomorphism h_i such that $h_i(body(\sigma_{i+1})) = head(\sigma_i)$; a sequence consisting of a single TGD is trivially tight. Furthermore, such a sequence is *compatible* to an atom $\underline{a}$ if there exists a homomorphism h such that $h(body(\sigma_1)) = \underline{a}$. We are now ready to introduce the central notion of *atom coverage*. For brevity, given an atom $\underline{a}$ and a term t, $pos(\underline{a}, t)$ is the set of positions at which t occurs in $\underline{a}$; for instance, if $\underline{a} = r(X, Y, X)$, then $pos(\underline{a}, X) = \{r[1], r[3]\}$ and $pos(\underline{a}, Y) = \{r[2]\}$. Moreover, given a CQ q and an atom $\underline{a} \in body(q)$, let $T(q, \underline{a})$ be the maximal subset of $terms(\underline{a})$ that contains only constants occurring in q and variables shared in q; for example, if q is the CQ $p(A) \leftarrow r(A, B, c)$ where $c \in \Gamma$, then $T(q, r(A, B, c)) = \{A, c\}$.

Definition 6.3 (*Atom Coverage*). Consider a CQ q over a schema $\mathcal{R}$ and a set $\Sigma \in$ LINEAR over $\mathcal{R}$. Let $\underline{a}$ and $\underline{b}$ be atoms of $body(q)$. We say that $\underline{a}$ *covers* $\underline{b}$ w.r.t. q and Σ, written as $\underline{a} \prec^q_\Sigma \underline{b}$, if the following conditions are satisfied:

(1) $T(q, \underline{b}) \subseteq terms(\underline{a})$, and
(2) there exists a sequence $S = \sigma_1, \ldots, \sigma_m$ of TGDs of Σ, for $m \geqslant 1$, such that:
   (a) S is tight and compatible to $\underline{a}$;
   (b) for each $t \in T(q, \underline{b})$ and $\pi \in pos(\underline{b}, t)$, there exists a minimal path $\pi_1 \pi_2 \ldots \pi_{m+1}$ in $PG(\Sigma)$ such that $\pi_1 \in pos(\underline{a}, t)$, $\pi_{m+1} = \pi$ and $\lambda(\langle \pi_j, \pi_{j+1} \rangle) = \sigma_j$, for each $j \in [m]$.

The *cover set* of an atom $\underline{a} \in body(q)$ w.r.t. q and Σ, denoted $cover(\underline{a}, q, \Sigma)$, is the set $\{\underline{b} \mid \underline{b} \in body(q) \setminus \{\underline{a}\}$ and $\underline{b} \prec^q_\Sigma \underline{a}\}$; when q and Σ are obvious from the context, we shall denote the aforesaid set as $cover(\underline{a})$.

Intuitively speaking, the first condition of atom coverage ensures that, by removing $\underline{b}$ from q we do not lose any constant and also that all the joins between $\underline{b}$ and the other atoms of $body(q)$, except $\underline{a}$, are preserved. The second condition guarantees that $\underline{b}$ is logically implied (w.r.t. Σ) by $\underline{a}$, thus can be safely eliminated. The choice of considering only minimal paths in condition 2(b) is crucial in order to be able to explicitly construct the cover set of an atom without considering infinite paths. Notice that, by considering infinite paths, we compute exactly the same cover sets. More precisely, if $\underline{a} \preccurlyeq^q_\Sigma \underline{b}$ denotes the fact that $\underline{a}$ covers $\underline{b}$ w.r.t. q and Σ if we consider infinite paths in Definition 6.3, then it is easy to verify that $\underline{a} \preccurlyeq^q_\Sigma \underline{b}$ implies $\underline{a} \prec^q_\Sigma \underline{b}$. In fact, if $\underline{a} \preccurlyeq^q_\Sigma \underline{b}$ because of a nonminimal path P, then we can construct a minimal path P' from P, by eliminating the repeated cycles, which is a witness for the fact that $\underline{a} \prec^q_\Sigma \underline{b}$.

LEMMA 6.4. *Consider a CQ q over a schema $\mathcal{R}$ and a set $\Sigma \in$ LINEAR over $\mathcal{R}$. Suppose that $\underline{a} \prec^q_\Sigma \underline{b}$, where $\{\underline{a}, \underline{b}\} \subseteq body(q)$, and q' is obtained from q by eliminating $\underline{b}$. Then, $q'(I) \subseteq q(I)$, for each instance I that satisfies Σ.*

PROOF. Fix a tuple of constants $\mathbf{t}$. Suppose there exists a homomorphism h such that $h(body(q')) \subseteq I$ and $h(\mathbf{V}) = \mathbf{t}$, where $\mathbf{V}$ are the distinguished variables of q'. We need to show that there exists a homomorphism $\hat{h}$ such that $\hat{h}(body(q)) \in I$ and $\hat{h}(\mathbf{V}) = \mathbf{t}$. Let us first give an auxiliary technical claim; its proof can be found in electronic Appendix C.1.

CLAIM 6.5. *There exists a linear TGD σ over $\mathcal{R}$ such that $\Sigma \models \sigma$, a substitution λ, and a substitution μ which is the identity on $var(body(q'))$, such that $\lambda(body(\sigma)) = \underline{a}$ and $\lambda(head(\sigma)) = \mu(\underline{b})$.*

Since $\underline{a} \in body(q')$, Claim 6.5 implies that $h(\lambda(body(\sigma))) \in I$. Recall that $\Sigma \models \sigma$ and thus $I \models \sigma$. This implies that there exists $h' \supseteq h|_{\mathbf{X}}$, where $\mathbf{X}$ are the variables that appear both in $\lambda(body(\sigma))$ and $\lambda(head(\sigma))$ such that $h'(\lambda(head(\sigma))) \in I$. Therefore $h'(\lambda(head(\sigma))) = h'(\mu(\underline{b}))$. Since μ is the identity on $var(body(q'))$, h and $h' \circ \mu$ are compatible. Consequently, the substitution $\rho = h \cup (h' \circ \mu)$ maps $body(q)$ to I and $\rho(\mathbf{V}) = h(\mathbf{V}) = \mathbf{t}$. The claim follows with $\hat{h} = \rho$. □

The preceding technical result provides the logical underpinning for the query elimination technique. More precisely, Lemma 6.4 suggests that, for each CQ q obtained by applying the rewriting step of XRewrite, those atoms of $body(q)$ that are logically implied (w.r.t. Σ) by some other atom of $body(q)$ can be eliminated and the obtained subquery is equivalent to q w.r.t. query answering.

Example 6.6. Consider the set Σ consisting of the linear TGDs

$$\sigma_1 : t(X,Y) \rightarrow \exists Z\, r(X,Y,Z), \quad \sigma_3 : s(X,Y,Z) \rightarrow t(Z,X),$$
$$\sigma_2 : r(X,Y,Z) \rightarrow \exists W\, s(Y,W,X), \quad \sigma_4 : t(X,Y) \rightarrow s(X,Y,Y).$$

Let also q be the CQ

$$p(A) \leftarrow \underbrace{t(A,B)}_{\underline{a}}, \underbrace{r(A,B,C)}_{\underline{b}}, \underbrace{s(A,B,B)}_{\underline{c}}.$$

By Definition 6.3, $cover(\underline{a}) = \{\underline{b}\}$, $cover(\underline{b}) = \{\underline{a}\}$ and $cover(\underline{c}) = \{\underline{a}, \underline{b}\}$. Thus, we can either eliminate $\underline{a}, \underline{c}$ and get the CQ $p(A) \leftarrow r(A,B,C)$ or eliminate $\underline{b}, \underline{c}$ and get the CQ $p(A) \leftarrow t(A,B)$. Both queries are equivalent to q (for query answering purposes).

6.3. Unique Elimination Strategy

The outcome of query elimination is not unique, as it heavily depends on the order in which we consider the atoms of the query under consideration. In the previous example, the order $\underline{a}, \underline{b}, \underline{c}$ gives the subquery $p(A) \leftarrow r(A,B,C)$, while the order $\underline{b}, \underline{a}, \underline{c}$ gives the subquery $p(A) \leftarrow t(A,B)$. Before presenting the optimized version of XRewrite, let us first discuss which elimination strategy best suits our needs.

ALGORITHM 3: The algorithm eliminate

Input: a CQ q, an elimination strategy S for q, and a set Σ of linear TGDs
Output: the set of eliminable atoms from $body(q)$ w.r.t. S and Σ

```
A := ∅;
foreach i := 1 to n do
    a := S[i];
    if cover(a, q, Σ) ≠ ∅ then
        A := A ∪ {a};
        foreach b ∈ body(q)\A do
            cover(b, q, Σ) := cover(b, q, Σ)\{a};
        end
    end
end
return A
```

An *(atom) elimination strategy* for a CQ q is a permutation of its body atoms. By exploiting the cover set of the atoms of $body(q)$, we associate to each elimination strategy S for q a subset of $body(q)$, denoted $\mathsf{eliminate}(q, S, \Sigma)$, namely the set of atoms of $body(q)$ that can be safely eliminated (according to S) in order to obtain a logically equivalent query (w.r.t. Σ) with less atoms in its body. Formally, $\mathsf{eliminate}(q, S, \Sigma)$ is computed by applying Algorithm 3; given an elimination strategy S, $S[i]$ is the i-th element of S. As already observed, given two strategies S_1 and S_2, in general, $\mathsf{eliminate}(q, S_1, \Sigma) \neq \mathsf{eliminate}(q, S_2, \Sigma)$. The question that comes up concerns the choice of the elimination strategy. Since our goal is to eliminate as many atoms as possible, we should choose an elimination strategy that maximizes the number of eliminable atoms. However, the process of finding such a strategy is computationally expensive. In particular, given a query with n body atoms, we have to enumerate the $n!$ different elimination strategies and, for each one of them, compute the set of eliminable atoms. Interestingly, such an expensive computation can be avoided since, regardless of the chosen elimination strategy, we always eliminate the same number of atoms. In other words, the strategy of eliminating atoms from the body of a query is unique (modulo the number of the eliminable atoms). The proof of this result (that can be found in electronic Appendix C.2) relies on the fact that the binary relation $\prec^q_\Sigma$ is transitive.

LEMMA 6.7. *Consider a CQ q and a set $\Sigma \in$ LINEAR. Let S_1 and S_2 be arbitrary elimination strategies for q. It holds that* $|\mathsf{eliminate}(q, S_1, \Sigma)| = |\mathsf{eliminate}(q, S_2, \Sigma)|$.

Henceforth, given a CQ q of the form $\underline{h} \leftarrow \underline{a}_1, \ldots, \underline{a}_n$, we refer to the atom elimination strategy for q, as denoted by S_q and we denote by $\lfloor q \rfloor_\Sigma$ the CQ obtained from q after eliminating from $body(q)$ the atoms of $\mathsf{eliminate}(q, S_q, \Sigma)$.

6.4. Query Elimination

We are now ready to describe the optimized algorithm XRewriteEliminate. During the execution of XRewrite, after the rewriting and factorization steps, the query elimination step is applied. XRewriteEliminate is obtained after modifying XRewrite as follows:

We have:

(1) line 2 — $Q_{\text{REW}} := \{\langle \lfloor q \rfloor_\Sigma, \mathsf{r}, \mathsf{u} \rangle\}$;
(2) line 10 — $q' := \lfloor \gamma_{S,\sigma^i}(q[S/body(\sigma^i)]) \rfloor_\Sigma$; and
(3) line 17 — $q' := \lfloor \gamma_S(q) \rfloor_\Sigma$.

Since eliminate terminates and XRewriteEliminate generates fewer queries than XRewrite, the termination of the optimized algorithm follows by Theorem 4.10.

THEOREM 6.8. *Consider a CQ q over a schema $\mathcal{R}$ and a set $\Sigma \in$ LINEAR over $\mathcal{R}$. Then,* $\mathsf{XRewriteEliminate}(q, \Sigma)$ *terminates.*

The next result establishes the correctness of XRewriteEliminate. For brevity, given a CQ q and a set Σ of linear TGDs, the query $\mathsf{XRewriteEliminate}(q, \Sigma)$ is denoted $q^\star_\Sigma$.

THEOREM 6.9. *Consider a CQ q over a schema $\mathcal{R}$, a database D for $\mathcal{R}$, and a set $\Sigma \in$ LINEAR over $\mathcal{R}$. It holds that $q^\star_\Sigma(D) = ans(q, D, \Sigma)$.*

PROOF. Since $D \subseteq chase(D, \Sigma)$, by monotonicity of CQs, $q^\star_\Sigma(D) \subseteq q^\star_\Sigma(chase(D, \Sigma))$. Thus $q^\star_\Sigma(D) \subseteq ans(q^\star_\Sigma, D, \Sigma)$. By giving a proof similar to that of Lemma 4.13 and also by exploiting Lemma 6.4, we can show that $ans(q^\star_\Sigma, D, \Sigma) \subseteq ans(\hat{q}, D, \Sigma)$, where $\hat{q} = \lfloor q \rfloor_\Sigma$. Since $chase(D, \Sigma) \models \Sigma$, Lemma 6.4 implies that $\hat{q}(chase(D, \Sigma)) \subseteq q(chase(D, \Sigma))$; hence, $ans(\hat{q}, D, \Sigma) \subseteq ans(q, D, \Sigma)$, thus implying $q^\star_\Sigma(D) \subseteq ans(q, D, \Sigma)$. Conversely, $body(\hat{q}) \subset body(q)$ implies $ans(q, D, \Sigma) \subseteq ans(\hat{q}, D, \Sigma)$. Since, by construction, $\hat{q} \in q^\star_\Sigma$, we immediately get that $ans(\hat{q}, D, \Sigma) \subseteq ans(q^\star_\Sigma, D, \Sigma)$. By devising a proof similar to

that of Lemma 4.14 and also by exploiting Lemma 6.4, we can show that $ans(q^\star_\Sigma, D, \Sigma) \subseteq q^\star_\Sigma(D)$. Therefore $ans(q, D, \Sigma) \subseteq q^\star_\Sigma(D)$ and the claim follows. □

It is important to clarify that the earlier result does not hold if we consider arbitrary TGDs. This is because Lemma 6.4, which is crucial in the proof of Theorem 6.9, is heavily based on the linearity of TGDs. Notice that the algorithm XRewriteEliminateParallel can be naturally defined by considering in the parallel step of XRewriteParallel the algorithm XRewriteEliminate instead of XRewrite.

6.5. The Chase-and-Backchase Approach

The task of finding all the minimal equivalent reformulations of a CQ with respect to a set of TGDs has been already investigated in databases. The most interesting approach in this respect is the Chase-and-Backchase (C&B) algorithm [Deutsch et al. 1999]. During the *chase phase*, the given CQ q is chased using the TGDs of the given set Σ, yielding a query q_U called *universal plan*. The *backchase phase* enumerates all minimal subqueries of q_U that are equivalent to q w.r.t. Σ; henceforth, we refer to Σ-minimal and Σ-equivalent subqueries. For a subquery q_S of q_U, to decide whether q_S is Σ-equivalent to q it suffices to check whether $q_S \subseteq_\Sigma q$, that is, whether q_S is contained in q w.r.t Σ, which reduces to finding a containment mapping from q to the query obtained after chasing q_S using Σ. Let us recall that, instead of naively enumerating all the possible subqueries of q_U during the backchase phase, one can employ a bottom-up approach starting with all subqueries with just one atom, continuing with those consisting of two atoms, and so on, and stopping as soon as a subquery that is Σ-equivalent to q is found. This is possible due to the so-called *pruning property*, which says that if a subquery q_S of q_U is Σ-equivalent to q, then every subquery of q_U that is a superquery of q_S cannot be both Σ-equivalent to q and Σ-minimal.

It is obvious that C&B is more general than our query elimination technique. More precisely, given a CQ q and a set Σ of linear TGDs, C&B will definitely return the CQ eliminate(q, S_q, Σ). Therefore, during the execution of XRewrite, the elimination of redundant atoms can be done by exploiting the C&B algorithm instead of relying on our query elimination technique. Unfortunately, C&B suffers two major drawbacks that make it inappropriate for our purposes. The first is the fact that it works only for classes of TGDs that guarantee the termination of the chase. Recall that in both phases of the algorithm we need to chase a query as long as no new atoms can be obtained. Thus, if we consider, for instance, arbitrary linear TGDs, then the termination of the procedure is not guaranteed. The second drawback (assuming that we focus on a class that guarantees the termination of the chase) is the fact that we need to apply the chase procedure double-exponentially many times (in general), which makes the whole procedure computationally expensive—recall that the main motivation underlying our backward-chaining algorithm was precisely the avoidance of the explicit construction of the chase. Therefore, although the C&B algorithm can be used to identify and eliminate redundant query atoms, the query elimination approach is more appropriate for our purposes, since it works for arbitrary linear TGDs and can effectively identify redundant atoms without an explicit construction of the chase.

7. IMPLEMENTATION

We implemented XRewrite and its optimizations in Java by extending the IRIS Datalog engine [Bishop and Fischer 2008]. Throughout this section we will refer to this implementation as IRIS$^\pm$. All data used in our evaluation, together with the complete source code of IRIS$^\pm$, are publicly available[9].

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

7.1. System Architecture

A high-level overview of the main architectural components of IRIS$^{\pm}$ and their interconnections is shown in Figure 5(a). The input of the system consists of a pair $\langle Q, \Sigma \rangle$: Q is a set of CQs to be executed against a (possibly incomplete) relational database D, while Σ is an ontology consisting of nonconflicting TGDs and FDs, as well as negative constraints (NCs)[10]. The IRIS$^{\pm}$ *parser* partitions Σ into Σ_T (the set of TGDs), Σ_F (the set of FDs), and $\Sigma_\perp$ (the set of NCs). The *constraints manager* accepts Σ_T and constructs (query-independent) support data structures based on Σ_T. In particular, the cover graph of Σ_T, which is basically the transitive closure of the propagation graph of Σ_T (see Definition 6.1), is constructed—more details are given in the following section. The constraints manager accepts also Σ_F and $\Sigma_\perp$ and then constructs a set Q_F and $Q_\perp$, respectively, of *check queries*, which are actually unions of CQs that will be used to verify whether D satisfies Σ_F and $D \cup \Sigma$ satisfies $\Sigma_\perp$. The *query manager* takes as input the set Q and schedules the CQs of Q for rewriting and execution.

Both input and check queries are handed over to the rewriting engine. More precisely, given as input a CQ $q \in Q$, the union of CQs $Q_\perp$, and the set of TGDs Σ_T (along with the cover graph of Σ_T), the rewriting engine rewrites q and $Q_\perp$ using XRewrite into a union of CQs Q_q and $\overline{Q}_\perp$, respectively. Then, the *SQL-Rewriter* accepts as input Q_q, Q_F, and $\overline{Q}_\perp$ and rewrites them into equivalent select-project-join SQL queries SQ_q, SQ_F, and $SQ_\perp$, respectively, to be executed against D. A nonempty answer to the (rewritten) check query SQ_F (respectively, $SQ_\perp$) implies that an FD of Σ_F (respectively, an NC of $\Sigma_\perp$) is violated, that is, $D \cup \Sigma$ is inconsistent. In this case, IRIS$^{\pm}$ exits with an error as well as list of violated constraints together with the tuples of D that "witness" the violation. If for the check queries the answer is the empty set, then IRIS$^{\pm}$ executes the rewritten query SQ_q over D.

Figure 5(b) shows in more detail the architectural structure of the IRIS$^{\pm}$ rewriting engine. The main module is the *FO-Rewriter* that implements XRewrite. The engine receives as input a CQ q and the set Σ_T along with the cover graph of Σ_T. First, it hands q and Σ_T over to the *query decomposer* that decomposes q into components $q_1, q_2, \ldots, q_k$, according to the procedure described in Section 5, that can be independently rewritten. The decomposer also computes the reconciliation rule ρ. Each q_i, where $i \in [k]$, is then handed over an independent FO-Rewriter that produces the rewriting Q_i for that particular component. Each atom of the reconciliation rule ρ is then unfolded using the corresponding rewriting Q_i. All FO-Rewriters share access to the graph $CG(\Sigma_T)$. Moreover, during the execution of the rewriting procedure, an additional data structure called *query graph* is maintained, that actually stores the CQs generated during the rewriting; more details are given in the following section. Furthermore, the FO-Rewriters share access to caching facilities that aim at avoiding to recompute several times the same piece of information, such as the MGU for a set of atoms that is needed for the execution of the rewriting process (discussed in more detail in Section 8.2). Notice that the *cache manager* ensures synchronized access to the caches.

It is worth noting that an indexing structure for TGDs is adopted. More precisely, *TGD-Index* is implemented as a map $M(K, V)$, where a key $k \in K$ is a predicate symbol of the underlying schema and where the value $M(k) \in V$ is a set of TGDs of Σ having k as head predicate. This allows an FO-Rewriter, during an applicability check, to consider only those TGDs that may be applicable. This is quite beneficial since, despite the fact that a single applicability check is computationally easy, the rewriting step iterates over each TGD $\sigma \in \Sigma$ checking applicability of σ to a set of atoms S in the

[10]As said in Section 4, our techniques apply even if we additionally consider a limited form of functional dependencies, called nonconflicting, and negative constraints; for details see electronic Appendix B.1.

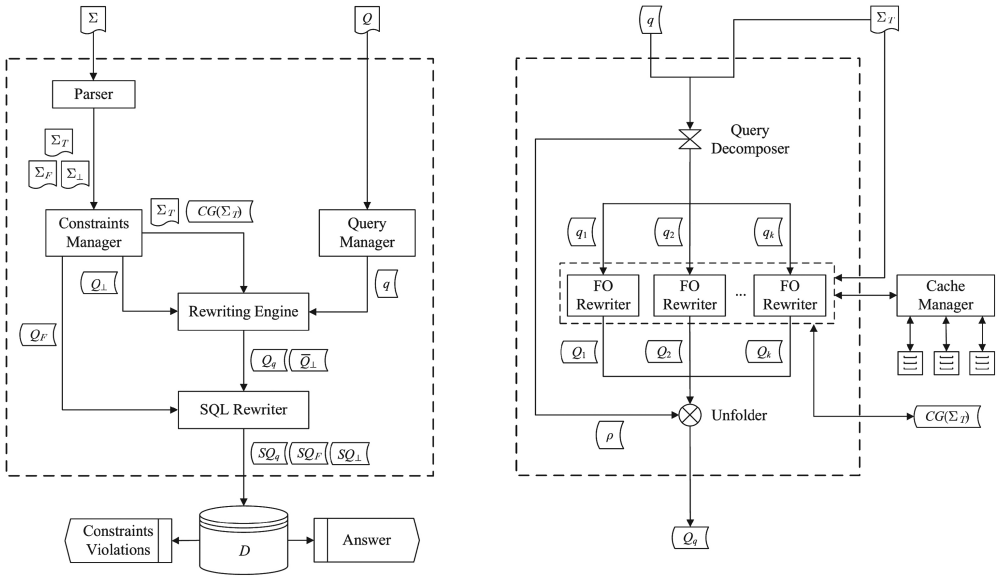

Fig. 5. IRIS$^{\pm}$ architecture components.

query q being rewritten and therefore, on large ontologies, this iteration might result in an unnecessary waste of time since only a few TGDs may be applicable to S.

7.2. Support Data Structures

As already said, IRIS$^{\pm}$ makes use of support data structures, namely the query graph and the cover graph. In what follows, we describe how these data structures are implemented as well as how they are used during the rewriting process.

Query Graph. The *query graph* stores the queries generated during the rewriting process. The formal definition follows.

Definition 7.1 (*Query Graph*). Consider a CQ q over a schema $\mathcal{R}$ and set Σ of TGDs over $\mathcal{R}$. The *query graph* of q and Σ is a labeled directed acyclic graph $\langle N, E, \lambda \rangle$, where N is the node set, E is the edge set, and λ is a labeling function $N \to \mathsf{XRewrite}(q, \Sigma)$. An edge $\langle v, u \rangle$ occurs in E if there exists $\sigma \in \Sigma$ and $S \subseteq \mathit{body}(q_v)$, where σ is applicable to S and $q_v = \lambda(v)$, as well as an integer $i \geqslant 1$, such that $\lambda(u) = \gamma_{S,\sigma^i}(q_v[S/\mathit{body}(\sigma^i)])$.

In other words, the preceding definition says that $\lambda(u)$ is obtained during the execution of $\mathsf{XRewrite}(q, \Sigma)$ by applying the rewriting step on $\lambda(v)$. Interestingly, apart from storing the generated queries, the query graph also keeps track of the provenance of the queries. This allows us, whenever a generated query is recognized as redundant because of a query subsumption check (that we are going to discuss in the next section), to use the edges in the graph to eventually remove its descendants, thus saving similar checks that are redundant. The query graph is implemented using JGraphT[11] that provides efficient data structures for the representation of graph-like structures and comes with efficient implementations of algorithms such as reachability.

[11] http://jgrapht.org.

Cover Graph. As said, the cover graph of a set Σ of TGDs is actually the transitive closure of the propagation graph of Σ. We denote by $\Sigma^\star$ the Kleene closure of Σ[12], namely the set of all strings over Σ of any length $n > 0$. By abuse of notation, if $PG(\Sigma) = \langle N, E, \lambda \rangle$ and $v_1 \dots v_n$ is a path in $PG(\Sigma)$, then by $\lambda(v_1 \dots v_n)$ we denote the string $\lambda(\langle v_1, v_2 \rangle)\lambda(\langle v_2, v_3 \rangle) \dots \lambda(\langle v_{n-1}, v_n \rangle) \in \Sigma^\star$. The formal definition follows.

Definition 7.2 (*Cover Graph*). Consider a set Σ of TGDs over a schema $\mathcal{R}$ and assume that $PG(\Sigma) = \langle N, E, \lambda \rangle$. The *cover graph* of Σ, denoted $CG(\Sigma)$, is a labeled directed multigraph $\langle N, E', \lambda' \rangle$, where $E' \supseteq E$ and $\lambda' : E' \to \Sigma^\star$. The edge set E' is defined as follows: *(i)* if there exists a minimal path $v_1 \dots v_n$, where $n > 1$, in $PG(\Sigma)$ such that the sequence of TGDs $\lambda(v_1 \dots v_n)$ is tight, then in E' there exists an edge $e = \langle v_1, v_n \rangle$ with $\lambda'(e) = \lambda(v_1 \dots v_n)$; and *(ii)* no other edges belong to E'.

The cover graph is used to check whether a certain position is reachable from some other position of the underlying schema. More precisely, it is used to check for the existence of a tight sequence of TGDs as required by the definition of atom coverage (see Definition 6.3). Moreover, it is used for the computation of the affected positions of the underlying schema (see Definition 5.1). Notice that in those cases where query elimination is not applied, such as when the input set of TGDs is not linear, and thus we do not need to check for atom coverage, then we can consider only the propagation graph (and not the cover graph). The cover graph is implemented as a map $M(K, V)$, where a key $k \in K$ is a pair $\langle \pi, \pi' \rangle$ of positions of the underlying schema such that π' is reachable from π via a sequence of TGDs and where the value $M(k) \in V$ is the set of all sequences s of TGDs such that π' is reachable from π via s. This implementation of the cover graph proved a better alternative than a traditional graph structure due to the potentially high number of calls to the reachability procedure; by precomputing the closure of the propagation graph, reachability can be checked in constant time.

8. EXPERIMENTAL EVALUATION

We are now ready to perform an experimental evaluation of XRewrite. After describing the experimental setting, we carried out an extensive internal evaluation in order to better understand the impact of the proposed optimization techniques. Finally, we compare our system with ALASKA, the reference implementation of König et al. [2012], the only known system supporting ontological query answering under general existential rules.

8.1. Experimental Setting

Since ontological query answering under existential rules is a relatively recent area of research, no benchmark is currently available. We therefore resorted to an established benchmark for DL-based query rewriting systems used, for instance, in Pérez-Urbina et al. [2010] and König et al. [2012]. The benchmark consists of five ontologies expressed in the well-known description logic DL-Lite$_\mathcal{R}$. Notice that every set of DL-Lite$_\mathcal{R}$ axioms can be translated into an equivalent set (with respect to query answering) of linear TGDs and NCs over a schema consisting of unary and binary predicates; for details, see Calì et al. [2012a]. A brief description of the ontologies follows.

—VICODI (V) is an ontology of European history developed within the VICODI project[13]. It consists of 222 linear TGDs without constraints.
—STOCKEXCHANGE (S) is an ontology of the domain of financial institutions within the EU. It consists of 53 linear TGDs without constraints.

[12]By abuse of notation, we consider Σ as a set of symbols.
[13]http://www.vicodi.org.

—UNIVERSITY (U) is a DL-Lite$_{\mathcal{R}}$ version of the LUBM benchmark[14] developed at Lehigh University that describes the organizational structure of universities. It consists of 87 linear TGDs without constraints.
—ADOLENA (A) (Abilities and Disabilities OntoLogy for ENhancing Accessibility) was developed for the South African National Accessibility Portal and describes abilities, disabilities, and devices. It consists of 154 linear TGDs and 19 NCs.
—The Path5 (P5) ontology is a synthetic ontology encoding graph structures and is used to generate an exponential blowup of the size of the rewritten queries. It consists of 13 linear TGDs without constraints.

Since XRewrite supports general existential rules, we have complemented the aforesaid benchmark with two ontologies consisting of linear and sticky sets of TGDs, respectively, that are not expressible using description logics.

—Split-Full (SF) is an ontology designed to test the ability of a rewriting algorithm to exploit query decomposition. It consists of 60 linear TGDs over a schema with predicates of arity at most three.
—Clique (CLQ) is an ontology representing k-cliques in a graph where $k \in [3]$ and has been devised to test the ability of rewriting engines to handle sticky sets of TGDs. It consists of 34 TGDs over a schema with predicates of arity at most four.

Each ontology has an associated set of test queries (see electronic Appendix D.1) either obtained via an analysis of query logs or manually created. Since XRewrite is provably sound and complete, we need some metrics for the quality of the rewriting.

—*Size.* When the target rewriting language is UCQs, the size represents the number of CQs in the final rewriting. Some existing approaches and systems, such as Orsi and Pieris [2011], Pérez-Urbina et al. [2010], and Rosati and Almatelli [2010], also support other languages for the rewriting such as nonrecursive or bounded Datalog. In this case the size of the rewriting is the number of rules in the Datalog program. Notice that the fact that Datalog rewritings are syntactically more succinct than UCQs does not immediately imply that they are preferable from a practical point of view. One of the reasons is the necessity to resort to Datalog engines or some form of preprocessing before being able to execute a Datalog rewriting against standard relational database systems. Other size-related metrics include the *number of joins* and the *number of atoms*, since they are an indication of the effort necessary to execute the rewriting. Since all disjuncts in the rewriting must be executed, in the following we always consider the total number of atoms and joins in the rewriting.
—*Rewriting time.* Assuming that the natural setting of ontological query answering is a transactional environment, another important metric is the time required to compute a final (and executable) rewriting once a query is submitted to the system. In this article, we do not include in this metric the time required for the construction of the cover graph, that does not depend on the query itself and can be constructed beforehand. However, we include query-dependent pre- and postprocessing steps such as query decomposition.
—*Memory consumption.* This represents the peak memory usage reached during the rewriting of a given query. This metric always includes the memory consumption introduced by caches but not that of auxiliary data structures such as the cover graph.
—*Search space.* Another typical metric for query rewriting algorithms is the number of CQs explored and generated during the rewriting [König et al. 2013]. In case of

[14]http://swat.cse.lehigh.edu/projects/lubm/.

XRewrite, the explored queries are those labelled with e, whereas the generated ones are those obtained via a rewriting step (possibly multiple times). Ideally, a rewriting algorithm should be able to explore and generate only those queries necessary for the final rewriting; as we shall see, this is not always the case.

The machine used for testing is a Dell Optiplex 9020 with 4 dual-core Intel i7-4770 processors at 3.40GHz (8 cores in total), running Linux Mint v15 (Olivia) x86-64, Kernel 3.8.0-19. The machine is equipped with 32GB of RAM. We used a Java VM 1.7.0-45 provided with 16GB of maximum heap size.

8.2. Caching Mechanism

During the rewriting process, several operations, such as the computation of the MGU for a set of atoms, are likely to be applied multiple times for the same input. This might occur either within a single FO-Rewriter, for example, because the same CQ is generated more than once in different branches of the rewriting procedure, or due to multiple FO-Rewriters exploring the same CQ in two different branches of the search space. For this reason, we have analyzed the behavior of XRewrite to identify operations that might benefit from caching. These operations are the following.

—*Query elimination.* Given a CQ q and a set Σ of TGDs, compute the query $\lfloor q \rfloor_\Sigma$.
—*MGU computation*. Given a set of atoms S, compute the MGU for S.
—*Canonical renaming.* Given a CQ q, compute the query $can_q(q)$.

Caches are implemented as maps $M(K, V)$, where the nature of keys and values varies depending on the particular cache; for details, see electronic Appendix D.2.

8.3. Internal Evaluation

The aim of the internal evaluation is to quantify the impact of our optimizations on the rewriting. In particular, they aim at: *(i)* reducing the number of redundant queries in the final rewriting while preserving its completeness, and *(ii)* intelligently exploring the rewriting search space, such as by avoiding the exploration of redundant queries.

Query Elimination. The first optimization we consider is query elimination (introduced in Section 6). Query elimination requires linearity of the TGDs, therefore we exclude the CLQ ontology from the analysis. Table I quantifies the gain produced by query elimination (QE) against a baseline (BASE), where XRewrite is run without applying any additional optimization steps (see Section 4.2).

Query elimination provides a substantial advantage in terms of the size of the rewriting for the ontologies U and S. In particular, for q_2 in U and S, all but one atom are eliminated from the input queries, thus resulting in a 98% reduction in the size of the rewriting. On the other side, query elimination is ineffective on V and P5. For the ontology V, the test queries as well as all the queries generated during the rewriting process are already "minimal" in the sense that no atoms are eliminated after applying query elimination. As a natural consequence, query elimination also has a beneficial effect on the exploration of the rewriting search space, since entire branches of the exploration space are pruned. This also impacts the running time and memory consumption. Again, a substantial improvement is observed on S and U, both in terms of explored and generated queries. For ontologies P5 and A we observe a gain in the exploration and generation of queries, although this does not translate to a substantially smaller size of the final rewriting. It is worth noting that, even when query elimination is less effective (i.e., A, P5, and V), the impact of the additional checks on the rewriting time and memory consumption is negligible.

Table I. Impact of Query Elimination on the Rewriting

		Size		#Atoms		#Joins		Explored		Generated		Time (ms)		Memory (MB)	
		BASE	QE	BASE	QE	BASE	QE	BASE	QE	BASE	QE	BASE	QE	BASE	QE
V	q_1	15	15	15	15	0	0	15	15	14	14	9	9	4.3	4.3
	q_2	10	10	30	30	30	30	10	10	9	9	7	7	4.3	6.3
	q_3	72	72	216	216	144	144	72	72	71	71	44	45	4.7	6.7
	q_4	185	185	555	555	370	370	185	185	184	184	111	115	5.4	7.4
	q_5	30	30	210	210	270	270	30	30	29	29	26	28	4.6	6.6
S	q_1	6	6	6	6	0	0	6	6	7	7	2	2	4.1	4.1
	q_2	160	2	480	2	320	0	160	2	244	1	43	2	5.9	8.2
	q_3	504	4	2,520	8	2,520	4	504	4	823	3	198	8	16.8	8.2
	q_4	960	4	4,800	8	4,800	4	960	4	1,445	3	363	2	25.3	8.2
	q_5	3,024	8	21,168	24	27,216	24	3,024	8	4,892	7	1.7s	3	12.6	8.3
U	q_1	2	2	4	4	2	2	5	5	4	4	3	3	4.1	6.2
	q_2	148	1	444	1	296	0	240	1	250	0	73	1	5.8	4.1
	q_3	224	4	1,344	16	2,016	20	1,008	12	1,007	11	432	7	18.5	8.3
	q_4	1,628	2	4,884	2	1,628	0	5,000	5	6,094	4	1.6s	3	54.1	8.2
	q_5	3,009	10	12,036	20	18,054	20	8,154	25	11,970	24	3.2s	8	119.2	8.4
A	q_1	402	299	779	573	377	274	782	679	847	725	818	729	7.9	17.0
	q_2	103	94	256	238	153	144	1,784	1,772	1,783	1,783	1.1s	1.2s	19.1	33.4
	q_3	104	104	520	520	520	520	4,752	4,752	4,751	4,751	3.2s	3.5s	62.7	97.5
	q_4	492	456	1,288	1,216	796	760	7,110	6,740	7,110	6,838	3.8s	3.5s	67.8	65.8
	q_5	624	624	3,120	3,120	3,120	3,120	76,122	69,448	76,121	70,457	52.3s	49.8s	1.1G	981.5
P5	q_1	6	6	6	6	0	0	14	14	13	13	1	2	4.1	4.1
	q_2	10	10	16	16	6	6	77	77	76	80	55	8	4.5	8.6
	q_3	13	13	29	29	16	16	410	400	409	413	57	52	7.4	11.4
	q_4	15	15	44	44	29	29	2,275	2,210	2,274	2,273	368	403	30.3	33.5
	q_5	16	16	60	60	44	44	13,522	13,085	13,521	13,424	3.2s	3.2s	211.7	208.3
SF	q_1	1	1	3	3	2	2	1	1	0	0	1	1	0.053	2.1
	q_2	125	125	375	375	250	250	125	125	124	124	30	33	5.1	7.1
	q_3	1,000	1,000	3,000	3,000	2,000	2,000	1,000	1,000	999	999	227	237	12.6	14.7
	q_4	8,000	8,000	24,000	24,000	16,000	16,000	8,000	8,000	7,999	7,999	2s	2.2s	77.2	77.0
	q_5	27,000	27,000	162,000	162,000	108,000	108,000	27,000	27,000	26,999	26,999	12.4s	12.4s	560.7	561.6

Parallelize the Rewriting. We now discuss how the decomposition-based parallelization of the rewriting procedure (Section 5) impacts the rewriting metrics. Differently from query elimination, parallelization is applicable regardless of the expressive power of the input ontology. Table II summarizes the results, where PARA denotes XRewrite with parallelization (and query elimination). The comparison is carried out against a baseline (BASE) where only query elimination is applied. Since the parallelization cannot reduce the final size of the rewriting, we report the size of the rewriting only to complement the results of Table I with the size of the rewriting for CLQ, where query elimination is not applied. The number of components (Comp), computed for each query and for each ontology, is also reported. As before, we also give the number of explored and generated CQs. Along with the overall rewriting time, we also report the time to rewrite all components (Rew), the time necessary to decompose the query under consideration (Split), and to unfold the rewritten components (Unfold). As usual, we also report the impact of the optimization on memory consumption.

An immediate conclusion is that, when the input query is decomposable, the rewriting search space can often be more efficiently explored. For certain ontologies such as SF and CLQ, the gain is substantial and also generally reflected into a lower rewriting time and memory consumption. For other ontologies such as V, even if the input query is

Table II. Impact of Parallelization on the Rewriting

			Size		Explored		Generated		Time (ms)					Memory (MB)	
		Comp	BASE	PARA	BASE	PARA	BASE	PARA	BASE	PARA	Rew	Split	Unfold	BASE	PARA
V	q_1	1	15	15	15	15	14	14	9	14	14	0	0	4.3	4.3
	q_2	3	10	10	10	12	9	9	7	4	3	1	0	6.3	6.4
	q_3	3	72	72	72	28	71	25	45	25	24	1	2	6.7	6.7
	q_4	3	185	185	185	43	184	40	115	26	26	0	3	7.5	7.4
	q_5	7	30	30	30	14	29	7	28	16	16	0	2	6.6	6.7
S	q_1	1	6	6	6	6	7	7	2	2	2	0	0	4.2	4.2
	q_2	1	2	2	2	2	1	1	2	2	1	0	0	8.2	8.2
	q_3	1	4	4	4	4	3	3	8	3	2	0	0	8.2	8.2
	q_4	2	4	4	4	4	3	2	2	3	2	0	0	8.2	8.2
	q_5	2	8	8	8	6	7	4	3	4	3	1	0	8.3	8.3
U	q_1	2	2	2	5	6	4	4	3	4	3	0	0	6.2	6.2
	q_2	1	1	1	1	1	0	0	1	1	1	0	0	4.1	4.1
	q_3	4	4	4	12	9	11	5	7	4	3	1	1	8.3	8.3
	q_4	1	2	2	5	5	4	4	3	3	2	0	0	8.2	8.2
	q_5	2	10	10	25	10	24	8	8	5	5	0	0	8.3	8.3
A	q_1	1	299	299	679	679	725	725	729	282	281	1	0	17.0	17.0
	q_2	1	94	94	1,772	1,772	1,783	1,783	1.2s	853	852	1	0	33.4	33.4
	q_3	3	104	104	4,752	4,754	4,751	4,751	3.5s	2.5s	2.5s	3	7	97.5	49.8
	q_4	1	456	456	6,740	6,740	6,838	6,838	3.5s	3.5s	3.5s	1	0	65.9	93.9
	q_5	2	624	624	69,448	69,449	70,457	70,486	49.8s	43.4s	43.4s	5	18	981.5	865.0
P5	q_1	1	6	6	14	14	13	13	2	2	1	0	0	4.1	4.1
	q_2	1	10	10	77	77	80	80	8	9	8	0	0	8.6	8.6
	q_3	1	13	13	400	400	413	413	52	61	61	0	0	11.4	11.4
	q_4	1	15	15	2,210	2,210	2,273	2,273	403	400	399	1	0	33.5	33.5
	q_5	1	16	16	13,085	13,085	13,424	13,424	3.2s	3.1s	3.1s	0	0	208.2	208.6
SF	q_1	3	1	1	1	3	0	0	1	3	2	1	1	2.1	6.2
	q_2	3	125	125	125	15	124	12	33	6	5	0	1	7.1	6.6
	q_3	3	1,000	1,000	1,000	30	999	27	237	15	14	1	10	14.7	9.1
	q_4	3	8,000	8,000	8,000	60	7,999	57	2.2s	82	82	0	73	770.4	274.4
	q_5	6	27,000	27,000	27,000	39	26,999	33	12.4s	472	471	1	464	561.6	121.4
CLQ	q_1	1	38	38	38	38	57	57	102	8	8	0	0	4.6	4.6
	q_2	2	38	38	38	39	54	56	140	15	14	1	1	4.6	4.7
	q_3	4	152	152	152	44	223	59	864	17	17	0	6	7.5	5.5
	q_4	5	5,776	5,776	5,776	82	9,871	112	48.3s	317	316	1	304	287.4	87.08

fully decomposable into atomic components, such as q_5, the decomposition could result in a loss of performance due to the overhead introduced by the multithreaded execution of FO-Rewriters. On the other hand, it is worth noting that this occurs for queries that can already be rewritten very quickly, even without applying query elimination. The results on the ontology A deserve further explanation. As can be seen, for both q_3 and q_5 the number of explored queries increases. The reason is that, for q_3 (respectively, q_5), two (respectively, one) of the computed components do not get rewritten and therefore count as two (respectively, one) additional explored queries, but no substantial gain is obtained from such a decomposition. This is not the case without decomposition, since they would have all be part of a unique query, counting as a single explored query. In addition, parallelization can potentially prevent applicability of query elimination if the covered and covering atoms reside in two different components. Another interesting observation is that the decomposition is more effective when the rewriting search space

can be partitioned into fairly similar subsets explorable by independently rewriting each component. This is the case, for instance, for q_5 on SF but not for q_3 and q_5 on A, where some components do not generate any rewriting. If we consider those tests where decomposition is more effective, such as SF and CLQ, we observe that most of the time is spent unfolding the rewritten components into a UCQ. A possible way of tackling this problem is to keep the rewriting "folded", namely as a nonrecursive Datalog rewriting; more details can be found in electronic Appendix D.3.

Query Subsumption. An common way of reducing the size of the rewriting is to check for queries that are subsumed by some other queries in the rewriting and eliminate them. Formally, given two CQs q_1 and q_2, we say that q_1 *subsumes* q_2 if there exists a homomorphism h such that $h(body(q_1)) \subseteq body(q_2)$ and $h(head(q_1)) = head(q_2)$. Let us clarify that such a (query) subsumption check is not explicitly included as part of XRewrite; it is a well-known technique that can be exploited by any rewriting algorithm. IRIS$^{\pm}$ implements query subsumption using three different modes. The first mode (TAIL) consists of applying an exhaustive subsumption check for each pair of queries in the final rewriting and by eliminating the subsumed ones together with all its descendants according to the query graph. The procedure preserves the subsumee in case it is a descendant of the subsumed query. This mode guarantees a minimal number of CQs in the final rewriting. The *intra-decomposition* mode (IDEC) applies the subsumption check at the end of the rewriting of a single component obtained after the decomposition of the input query. This mode has the advantage that the subsumption check is applied on smaller queries and on smaller rewriting sets; however, it does not guarantee minimality of the final rewriting since a redundant query may be obtained during the unfolding step. Note that, if the query is not decomposable, then IDEC coincides with TAIL. The *intra-rewriting* mode (IREW) applies the subsumption check every time a new query is generated by a rewriting step. This mode has the advantage of shrinking the rewriting search space by pruning redundant CQs as soon as they are generated, but has the disadvantage that it might prevent completeness. As for IDEC, if a query is decomposable, then IREW does not guarantee minimality; otherwise, IREW coincides with TAIL.

Table III reports on the impact of the preceding three modes on the final rewriting. The comparison is carried out against a baseline (BASE) where query elimination and parallelization are applied. Notice that the number of explored and generated CQs is reported only for IREW, since it is the only subsumption check mode that has a potential effect on the exploration of the rewriting search space. The last two groups of columns report on the effect of the different subsumption check modes on the rewriting time and memory consumption. The symbol "†" denotes that the rewriting did not terminate within 15 minutes.

A first interesting observation is that the baseline algorithm already computes a minimal rewriting in most of the cases, with the exception of queries q_1, q_2, and q_4 on A. Also, the number of explored and generated queries matches those explored and generated by the intra-rewriting subsumption check for all queries in S, U, SF, and CLQ. On the other hand, IREW adds a substantial burden in terms of rewriting time, becoming impractical for q_5 on A and P5 where our algorithm does not terminate within 15 minutes from its invocation. Another observation is that, despite the fact that only TAIL provably guarantees the minimality of the rewriting, both IDEC and IREW produce a minimal number of CQs for the given input queries and ontologies. Also, TAIL becomes impractical for complex queries on SF and CLQ, whereas both IDEC and IREW terminate with timings comparable to the baseline.

In summary, our tests indicate that IDEC provides a good trade-off between the need for minimization of the rewriting and performance. Also, it seems that the amount of

Table III. Impact of Subsumption Check on the Rewriting

		Size				Explored		Generated		Time (ms)				Memory (MB)			
		BASE	TAIL	IDEC	IREW	BASE	IREW	BASE	IREW	BASE	TAIL	IDEC	IREW	BASE	TAIL	IDEC	IREW
V	q_1	15	15	15	15	15	15	14	14	14	12	13	10	4.3	4.3	4.3	4.3
	q_2	10	10	10	10	12	12	9	9	4	9	7	26	6.4	6.4	6.4	6.4
	q_3	72	72	72	72	28	28	25	25	25	28	15	24	6.6	6.6	6.6	6.6
	q_4	185	185	185	185	43	43	40	40	26	75	28	55	7.4	7.4	7.4	7.4
	q_5	30	30	30	30	14	14	7	7	16	12	11	14	6.7	6.7	6.7	6.7
S	q_1	6	6	6	6	6	6	7	7	2	2	3	3	4.2	4.2	4.2	4.2
	q_2	2	2	2	2	2	2	1	1	2	2	3	2	8.2	8.2	8.2	8.2
	q_3	4	4	4	4	4	4	3	3	3	3	3	2	8.3	8.3	8.3	8.3
	q_4	4	4	4	4	4	4	2	2	3	3	4	5	8.3	8.3	8.3	8.3
	q_5	8	8	8	8	6	6	4	4	4	3	6	5	8.3	8.3	8.3	8.3
U	q_1	2	2	2	2	6	6	4	4	4	4	6	5	6.2	6.2	6.2	6.2
	q_2	1	1	1	1	1	1	0	0	1	1	1	2	4.2	4.2	4.2	4.2
	q_3	4	4	4	4	9	9	5	5	4	4	3	5	8.3	8.3	8.3	8.3
	q_4	2	2	2	2	5	5	4	4	3	2	4	3	8.3	8.3	8.3	8.3
	q_5	10	10	10	10	10	10	8	8	5	3	6	5	8.3	8.3	8.3	8.3
A	q_1	299	27	27	27	679	41	725	45	282	771	325	168	17.0	12.0	17.0	8.7
	q_2	94	50	50	50	1,772	1,431	1,783	1,456	853	1.2s	917	15s	33.5	33.7	33.2	32.3
	q_3	104	104	104	104	4,754	4,468	4,751	4,467	2.5s	2.8s	2.5s	2m	49.9	50.0	46.3	43.2
	q_4	456	224	224	224	6,740	3,159	6,838	3,410	3.4s	3.6s	3.4s	1.3m	93.9	111.9	97.5	50.2
	q_5	624	624	624	624	69,449	32,922	70,486	38,902	43.4s	44.5s	43.2s	†	865.1	859.7	863.9	†
P5	q_1	6	6	6	6	14	14	13	13	2	2	3	2	4.2	4.2	4.2	4.2
	q_2	10	10	10	10	77	25	80	47	9	9	11	15	8.6	8.6	8.6	8.6
	q_3	13	13	13	13	400	60	413	208	61	52	53	303	11.4	11.4	11.4	14.95
	q_4	15	15	15	15	2210	180	2273	936	400	391	375	11s	33.5	33.5	33.5	124.2
	q_5	16	16	16	16	13085	725	13424	5188	3s	3.1s	3.3s	†	208.6	208.4	208.3	†
SF	q_1	1	1	1	1	3	3	0	0	3	30	3	5	6.2	6.2	6.2	6.2
	q_2	125	125	125	125	15	15	12	12	6	97	11	11	6.6	6.6	6.6	6.6
	q_3	1,000	1,000	1,000	1,000	30	30	27	27	15	1.5s	23	28	9.1	6.6	9.1	9.1
	q_4	8,000	8,000	8,000	8,000	60	60	57	57	82	83s	84	89	27.4	83.6	27.4	27.4
	q_5	27,000	27,000	27,000	27,000	39	39	33	33	472	†	427	415	121.5	384.5	121.5	†
CLQ	q_1	38	38	38	38	38	38	57	57	8	41	28	45	4.6	5.1	5.1	5.1
	q_2	38	38	38	38	39	39	56	56	15	41	31	60	4.7	5.3	5.3	5.3
	q_3	152	152	152	152	44	44	59	59	17	1.3s	38	56	5.5	17.7	6.0	6.0
	q_4	5,776	5,776	5,776	5,776	82	82	112	112	317	†	41	426	87.1	†	88.1	88.1

resources necessary to remove redundant queries via TAIL or IREW is not justified by the gain in size, especially if we consider caching mechanisms at the database level.

8.4. Computing the Support Data Structures

XRewrite relies on a number of data structures, namely query, propagation, and cover graphs, supporting the rewriting process. A natural question is how large such data structures can be and how long does it take to compute them. For the query graph the answer to such questions is straightforward, since its maximum size corresponds to the number of queries generated by XRewrite when no subsumption check is applied. Similarly, the time to compute it and the memory consumption roughly correspond to the rewriting time and the total memory usage of XRewrite.

Differently from the query graph, the propagation and the cover graph depend only on the input ontology and not on the input query. Table IV reports the characteristics of both the propagation graph (P-GRAPH) and the cover graph (C-GRAPH) constructed for

Table IV. Propagation and Cover Graphs

	Size (#nodes,#edges)		LP	Time (ms)		Memory	
	P-GRAPH	C-GRAPH	C-GRAPH	P-GRAPH	C-GRAPH	P-GRAPH	C-GRAPH
V	(214,445)	(214,1194)	7	4	158	190Kb	4.7Mb
S	(41,103)	(41,405)	8	1	120	45Kb	4,4Mb
U	(86,189)	(86,416)	6	1	37	81Kb	4.4Mb
A	(135,319)	(135,1133)	10	43	708	154Kb	4.7Mb
P5	(15,32)	(15,43)	3	0	1	14Kb	4.3Mb
SF	(100,195)	(100,1,050)	19	1	346	84Kb	4.8Mb
CLQ	(11,143)	N/A	N/A	2	N/A	39Kb	N/A

each ontology. In particular, we report on the size of the two structures in terms of the number of nodes and edges, the time necessary to construct them, and their memory footprint. For the cover graph, we also report the length of the longest label on an edge (LP), corresponding to the longest tight sequence of TGDs that we have to consider during the computation of cover sets. Since query elimination can be applied only to linear TGDs, for the sticky ontology CLQ no cover graph is computed.

Apart from S and V, in all other cases the time to compute the cover graph is either negligible or comparable to the time to rewrite a query w.r.t. the corresponding ontology. For S and V, the reason for the higher cost compared with the time necessary to rewrite the input queries lies in the fact that these ontologies are relatively simple and most of the machinery devised for the general case is not needed to efficiently handle these cases. On the other hand, considering the improvements that these two structures bring in terms of rewriting size, rewriting time, and memory consumption for the general case, it is certainly worthwhile to make use of them.

8.5. Comparative Evaluation

Although several DL-based systems exist that can deal with the DL-Lite$_{\mathcal{R}}$ ontologies in our tests, to the best of our knowledge only ALASKA (i.e., the reference implementation of König et al. [2012]) supports ontological query answering under general TGDs. We believe that limiting the comparison to these two systems is fair. DL-based systems leverage specificities of DLs, such as the limitation to unary and binary relations only and the absence of variable permutations in the axioms, that enable more efficient rewriting techniques not easily extended to more general languages such as TGDs; in fact, DL-based systems often resort to case-by-case analysis on the syntactic form of DL axioms. In addition to the queries provided by the benchmarks, we also generated 492 additional queries using SYGENIA [Imprialou et al. 2012], an automatic query generation tool for testing the completeness of rewriting-based DL systems. These queries do not cover the non-DL ontologies SF and CLQ. For space reasons, Table V limits the results of the evaluation to the benchmark queries. Results for the full (internal and comparative) evaluation are available online.[15]

For ALASKA we chose the setting that consistently reported the smallest size of the rewriting and, in case of a tie, that with lower rewriting time, namely `ar-single` in ALASKA terminology. In case of IRIS$^{\pm}$, we apply query elimination, parallelization, and an intra-decomposition subsumption check.

Transient states of the experimental machines can bias running time and memory consumption values. For a fair comparison, we run both systems 10 times and report the median of the values to limit bias due to outliers. Also, since code instrumentation for running time can interfere with memory consumption values and vice versa, 10 runs

[15]https://bitbucket.org/giorsi/nyaya/src.

Table V. ALASKA vs. IRIS$^{\pm}$

		Size		Explored		Generated		Time (ms)			Memory (MB)	
		ALASKA	IRIS$^{\pm}$	ALASKA	IRIS$^{\pm}$	ALASKA	IRIS$^{\pm}$	ALASKA	IRIS$^{\pm}$	S	ALASKA	IRIS$^{\pm}$
V	q_1	15	15	15	15	14	14	116	13	✓	.024	4.3
	q_2	10	10	10	12	9	9	19	11	×	.024	6.4
	q_3	72	72	72	28	117	25	36	21	✓	.054	6.7
	q_4	185	185	185	43	328	40	60	37	✓	.69	7.4
	q_5	30	30	30	14	59	7	5	13	×	.174	6.7
S	q_1	6	6	6	6	9	7	0	2	✓	.039	4.2
	q_2	2	2	48	2	288	1	7	2	✓	.004	8.2
	q_3	4	4	54	4	686	3	25	3	✓	.002	8.3
	q_4	4	4	192	4	1,632	2	56	4	✓	.005	8.3
	q_5	8	8	224	6	3,424	4	195	5	✓	.013	8.3
U	q_1	2	2	5	6	4	4	23	6	✓	.011	6.3
	q_2	1	1	42	1	148	0	119	2	✓	.002	4.2
	q_3	4	4	48	9	260	5	82	5	✓	.001	8.3
	q_4	2	2	1,300	5	6,092	4	2.4s	4	✓	.006	8.3
	q_5	10	10	100	10	1,430	8	233	5	✓	.003	8.3
A	q_1	27	27	457	679	1,307	725	517	324	✓	16	17.0
	q_2	50	50	1,598	1,772	4,658	4,704	2s	1.21s	✓	.050	17.85
	q_3	104	104	4,477	4,754	1,3871	4,751	4.5s	2.5s	✓	.697	46.6
	q_4	224	224	4,611	6,740	15,889	6,838	3.8s	3.5s	✓	.716	97.7
	q_5	624	624	50,508	69,449	231,899	70,486	12.8m	42.4s	✓	3.5	863.9
P5	q_1	6	6	14	14	13	13	0	2	✓	.004	4.2
	q_2	10	10	67	77	130	80	4	9	✓	.007	8.6
	q_3	13	13	332	400	1,001	413	74	50	×	.010	11.4
	q_4	15	15	1,647	2,210	7,065	2,273	2.6s	378	✓	3.5	33.5
	q_5	16	16	8,186	13,085	47,608	13,424	2m	3s	✓	.914	208.3
SF	q_1	1	1	1	3	0	0	0	3.5	✓	1	6.2
	q_2	125	125	125	15	300	12	10	7	×	122	6.6
	q_3	1,000	1,000	1,000	30	2,800	27	193	19	✓	973	9.1
	q_4	8,000	8,000	8,000	60	23,600	57	7.1s	93	✓	6.4	27.4
	q_5	27,000	27,000	27,000	39	135,000	33	3.6m	425	✓	40.0	121.5
CLQ	q_1	38	38	38	38	218	57	23	25	✓	37	5.1
	q_2	38	38	38	39	218	56	65	32	✓	41	5.3
	q_3	152	152	152	44	1,452	59	1.3s	36	✓	193	6.0
	q_4	5,776	5,776	82	82	112	112	†	346	-	†	48.8

have been performed only with code instrumented for running time and another 10 with code instrumented for memory consumption. Moreover, the column (S) shows whether the difference in running time between ALASKA and IRIS$^{\pm}$ is statistically significant (✓) or not (×). For a query q, we say that the difference in running time is significant if it is greater than the maximum standard deviation recorded for q on the two systems, that is, if $|time(q, \text{ALASKA}) - time(q, \text{IRIS}^{\pm})| > \max\{f(q, \text{ALASKA}),\ f(q, \text{IRIS}^{\pm})\}$, where $time(q, s)$ is the rewriting time for q on system s and where $f(q, s)$ denotes the standard deviation recorded for q on s over the 10 runs. As before, the symbol "†" denotes test cases where the rewriting process either did not terminate within 15 minutes or ran out of memory. A value of 0 indicates a running time below the millisecond.

A first observation is that both systems return minimal UCQ rewritings on the given test cases. A second is that query elimination allows IRIS$^{\pm}$ to perform a better exploration of the rewriting search space on V, S, and U, where it is more effective,

whereas ALASKA explores the search space better on A and P5. This is due to the better normalization of TGDs with multiple heads applied by ALASKA that we are planning to consider also for IRIS$^{\pm}$. On the other hand, on these ontologies, caching allows IRIS$^{\pm}$ to perform better than ALASKA since both query elimination and parallelization are rather ineffective on these ontologies. On SF and CLQ, parallelization provides a fundamental contribution towards making the rewriting manageable, as the number of explored and generated queries is drastically reduced. As expected, ALASKA consumes less memory and delivers better performance than IRIS$^{\pm}$ on simpler queries.

By extending the comparison to the full set of SYGENIA-generated queries, the following facts can be observed. All generated queries have length (i.e., number of atoms) less than 3 and are therefore considerably simpler than those provided by the benchmark. This is due to the fact that SYGENIA's goal is to test for completeness and not meant to stress-test the rewriting engines. On 80% of the test queries, IRIS$^{\pm}$ generates a rewriting of the same size as ALASKA, while for the remaining 20%, ALASKA produces smaller rewritings. This is attributable to the parallelization that prevents subsumption check across components. By running IRIS$^{\pm}$ with a TAIL subsumption check, it can be verified that the outputs of ALASKA and of IRIS$^{\pm}$ coincide in size for all queries. In terms of exploration and generation of queries, IRIS$^{\pm}$ explores and generates fewer queries than ALASKA in 78% of the cases, whereas ALASKA explores the search space better in 22% of the cases. This is again due to the parallelization that prevents atom coverage from identifying redundant atoms across different components.

9. CONCLUSIONS

The problem of designing a practical query rewriting algorithm for arbitrary TGDs has been investigated. A resolution-based query rewriting algorithm, called XRewrite, for linear and sticky TGDs has been proposed and several optimization techniques have been studied. An extensive analysis on the impact of the proposed optimizations on the rewriting process, as well as a comparison of our system with the only known system supporting query rewriting under arbitrary TGDs, that is, ALASKA, has also been performed. In the future, we would like to study in more depth the problem of parallelizing the rewriting process. In particular, we are planning to investigate more sophisticated techniques of decomposing the input query into smaller queries that can be independently rewritten. Also, effective execution of large rewritings in the form of UCQs as well as Datalog rewritings will be investigated.

ACKNOWLEDGMENTS

We thank Michaël Thomazo for his useful and constructive comments on the conference version of this article.

REFERENCES

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

Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. 2005. QuOnto: Querying ontologies. In *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference*. 1670–1671.

Miklos Ajtai and Yuri Gurevich. 1989. Datalog vs. first-order logic. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. 142–147.

Hajnal Andreka, Johan van Benthem, and Istvan Nemeti. 1998. Modal languages and bounded fragments of predicate logic. *J. Philos. Logic* 27, 217–274.

Franz Baader. 2003. Least common subsumers and most specific concepts in a description logic with existential restrictions and terminological cycles. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. 319–324.

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

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

Catriel Beeri and Moshe Y. Vardi. 1981. The implication problem for data dependencies. In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming*. 73–85.

Michael Benedikt, Balder Ten Cate, and Efthymia Tsamoura. 2014. Generating low-cost plans from proofs. In *Proceedings of the 33rd ACM Symposium on Principles of Database Systems*. 200–211.

Barry Bishop and Florian Fischer. 2008. IRIS - Integrated rule inference system. In *Proceedings of the International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense.*

Philip Bohannon, Eiman Elnahrawy, Wenfei Fan, and Michael Flaster. 2006. Putting context into schema matching. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 307–318.

Loreto Bravo, Wenfei Fan, and Shuai Ma. 2007. Extending dependencies with conditions. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 243–254.

Andrea Cali, Domenico Lembo, and Riccardo Rosati. 2003. Decidability and complexity of query answering over incosistent and incomplete databases. In *Proceedings of the 22nd ACM Symposium on Principles of Database Systems*. 260–271.

Andrea Cali, Georg Gottlob, and Michael Kifer. 2008. Taming the infinite chase: Query answering under expressive relational constraints. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning*. 70–80.

Andrea Cali, Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris. 2011. A logical toolbox for ontological reasoning. *SIGMOD Rec.* 40, 3, 5–14.

Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. 2012a. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Semant.* 14, 57–83.

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

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

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. Autom. Reason.* 39, 3, 385–429.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2013a. Data complexity of query answering in description logics. *Artif. Intell.* 195, 335–360.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2013b. Data complexity of query answering in description logics. *Artif. Intell.* 195, 335–360.

Ashok K. Chandra and Philip M. Merlin. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*. 77–90.

Ashok K. Chandra and Moshe Y. Vardi. 1985. The implication problem for functional and inclusion dependencies. *SIAM J. Comput.* 14, 671–677.

Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theor. Comput. Sci.* 239, 2, 211–229.

Alexandros Chortaras, Despoina Trivela, and Giorgos B. Stamou. 2011. Optimized query rewriting for owl 2 ql. In *Proceedings of the 23rd International Conference on Automated Deduction*. 192–206.

Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *VLDB J.* 22, 1, 73–98.

Alin Deutsch, Alan Nash, and Jeff B. Remmel. 2008. The chase revisisted. In *Proceedings of the 27th ACM Symposium on Principles of Database Systems*. 149–158.

Alin Deutsch, Lucian Popa, and Val Tannen. 1999. Physical data independence, constraints, and optimization with universal plans. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 459–470.

Alin Deutsch and Val Tannen. 2003. MARS: A system for publishing xml from mixed and redundant storage. In *Proceedings of the 29th International Conference on Very Large Data Bases*. 201–212.

Ronald Fagin, Phokion G. Kolaitis, Renee J. Miller, and Lucian Popa. 2005. Data exchange: Semantics and query answering. *Theor. Comput. Sci.* 336, 1, 89–124.

Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. 1993. Undecidable optimization problems for database logic programs. *J. ACM* 40, 3, 683–713.

Georg Gottlob, Stanislav Kikot, Roman Kontchakov, Vladimir Podolskii, Thomas Schwentick, and Michael Zakharyaschev. 2014. The price of query rewriting in ontology-based data access. *Artif. Intell.* 213, 42–59.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.* 64, 3, 579–627.

Georg Gottlob, Giorgio Orsi, and Andreas Pieris. 2011. Ontological queries: Rewriting and optimization. In *Proceedings of the 27th International Conference on Data Engineering*. 2–13.

Georg Gottlob and Thomas Schwentick. 2012. Rewriting ontological queries into small nonrecursive datalog programs. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning.*

Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4, 270–294.

Martha Imprialou, Giorgos Stoilos, and Bernardo Cuenca Grau. 2012. Benchmarking ontology-based query rewriting systems. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*. 779–785.

David S. Johnson and Anthony C. Klug. 1984. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.* 28, 1, 167–189.

Stanislav Kikot, Roman Kontchakov, and Michael Zakharyaschev. 2012a. Conjunctive query answering with OWL 2 QL. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning*.

Stanislav Kikot, Roman Kontchakov, Vladimir V. Podolskii, and Michael Zakharyaschev. 2012b. Exponential lower bounds and separation for query rewriting. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming*. 263–274.

Melanie Konig, Michel LeClere, Marie-Laure Mugnier, and Michael Thomazo. 2012. A sound and complete backward chaining algorithm for existential rules. In *Proceedings of the 6th International Conference on Web Reasoning and Rule Systems*. 122–138.

Melanie Konig, Michel LeClere, Marie-Laure Mugnier, and Michael Thomazo. 2013. On the exploration of the query rewriting space with existential rules. In *Proceedings of the 7th International Conference on Web Reasoning and Rule Systems*. 123–137.

Markus Krotzsch and Sebastian Rudolph. 2011. Extending decidable existential rules by joining acyclicity and guardedness. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. 963–968.

David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst.* 4, 4, 455–469.

Giorgio Orsi and Andreas Pieris. 2011. Optimizing query answering under ontological constraints. *Proc. VLDB Endow.* 4, 11, 1004–1015.

Christos. H. Papadimitriou. 1994. *Computational Complexity*. Addison-Wesley.

Hector Perez-Urbina, Boris Motik, and Ian Horrocks. 2010. Tractable query answering and rewriting under description logic constraints. *J. Appl. Logic* 8, 2, 186–209.

Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. 2008. Linking data to ontologies. *J. Data Semant.* 10, 133–173.

Mariano Rodriguez-Muro and Diego Calvanese. 2012. Quest, an OWL 2 QL reasoner for ontology-based data access. In *Proceedings of the OWL: Experiences and Directions Workshop*.

Riccardo Rosati and Alessandro Almatelli. 2010. Improving query answering over dl-lite ontologies. In *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning.*

Sebastian Rudolph, Markus Krotzsch, and Pascal Hitzler. 2008. All elephants are bigger than all mice. In *Proceedings of the 21st International Workshop on Description Logics*.

Balder Ten Cate and Phokion G. Kolaitis. 2009. Structural characterizations of schema-mapping languages. In *Proceedings of the 12th International Conference on Database Theory*. 63–72.

Michael Thomazo. 2013. Compact rewritings for existential rules. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence.*

Moshe Y. Vardi. 1995. On the complexity of bounded-variable queries. In *Proceedings of the 14th ACM Symposium on Principles of Database Systems.* 266–276.

Tassos Venetis, Giorgos Stoilos, and Giorgos Stamou. 2013. Query extensions and incremental query rewriting for OWL 2 QL ontologies. *J. Data Semant.* 3, 1, 1–23.

Received October 2013; revised April 2014; accepted June 2014