Ontological Queries: Rewriting and Optimization

Georg Gottlob\textsuperscript{1,2}, Giorgio Orsi\textsuperscript{1,3}, Andreas Pieris\textsuperscript{1}

\textsuperscript{1}Computing Laboratory, University of Oxford, UK
\textsuperscript{2}Oxford-Man Institute of Quantitative Finance, University of Oxford, UK
\textsuperscript{3}Dip. di Elettronica e Informazione, Politecnico di Milano, Italy

\{georg.gottlob,giorgio.orsi,andreas.pieris\}@comlab.ox.ac.uk

Abstract—Ontological queries are evaluated against an enterprise ontology rather than directly on a database. The evaluation and optimization of such queries is an intriguing new problem for database research. In this paper 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 query against the underlying relational database. The focus here is on soundness and completeness. We review previous results and present a new rewriting algorithm for rather general types of ontological constraints (description logics). In particular, we show how a conjunctive query (CQ) against an enterprise ontology can be compiled into a union of conjunctive queries (UCQ) against the underlying database. Ontological query optimization, in this context, attempts to improve this process so to produce possibly small and cost-effective output UCQ. We review existing optimization methods, and propose an effective new method that works for Linear Datalog\textsuperscript{1}, a description logic that encompasses well-known description logics of the DL-Lite family.

I. INTRODUCTION

This paper is about ontological query processing, an important new challenge to database research. We will review existing methods and propose new algorithms for compiling an ontological query, that is, a query against an enterprise ontology on top of a relational database, into a direct query against this database, and we will deal with optimization issues related to this process so as to obtain possibly small and efficient translated queries. In this section, we first discuss a number of relevant concepts, and then illustrate query rewriting and optimization processes in the context of a small but non-trivial example.

Ontologies. The use of ontologies and 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 \cite{1}, and consists of a formal representation of knowledge as a set of concepts within a domain, and the relationships between those concepts \cite{2}. To distinguish an enterprise ontology from a data dictionary, Dave McComb explicitly refers to the formal semantics of ontologies that enables automated processing and inferencing, while the interpretation of a data dictionary is strictly done by humans \cite{3}. 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.

Due to their high expressive power, ontologies are also substituting more traditional conceptual models such as UML class-diagrams and E/R schemata.

Description Logics. Description logics (DLs) are logical languages for expressing and modelling ontologies. The best known DLs are those underlying the OWL language\textsuperscript{1}. The main ontological reasoning and query answering tasks in the complete OWL language, called OWL Full, are undecidable. For the most well-known decidable fragments of OWL, ontological reasoning and query answering is still computationally very hard, typically 2\textsuperscript{EXPTIME}-complete.

In description logics, the ontological axioms are usually divided into two sets: The A-Box (assertional box), which essentially contains factual knowledge such as “IBM is a company”, denoted by company(ibm), or “IBM is listed on the NASDAQ”, which could be represented as a fact of the form list\_comp(ibm, nasdaq), and a T-Box (terminological box) which contains axioms and constraints that allow us, on the one hand, to infer new facts from those given in the A-box, and, on the other hand, to express restrictions such as keys. For example, a T-Box may contain an axiom stating that for each fact list\_comp(X, Y), Y must be a financial index, which in DL is expressed as ∃list\_comp ⊑ fin\_idx. If the fact fin\_idx(nasdaq) is not already present in the A-Box, it can be derived via the above axiom from list\_comp(ibm, nasdaq). Thus, the atomic query “q(X) ← fin\_idx(X)” would return nasdaq as one of the answers. Note that the axiom ∃list\_comp ⊑ fin\_idx, which corresponds to an inclusion dependency, is enforced by adding new tuples, rather than just being checked. This is one main difference between ontological constraints and classical database dependencies. In database terms, the above axiom is to be interpreted more like a trigger than a classical constraint.

Ontology-Based Data Access (OBDA). We are currently witnessing the marriage of ontological reasoning and database technology. In fact, this amalgamation consists in the realization of the obvious idea that A-Boxes shall be implemented in form of a relational database, or even stored in classical RDBMSs. Moreover, very large existing databases are semantically enriched with ontological constraints. There are a number of recent commercial systems and experimental

\textsuperscript{1}http://www.w3.org/TR/owl2-overview/
prototypes that extend relational DBMSs with the possibility of querying an ontology that is rooted in a database (for examples, see Section II). The main problem here is how to couple these two different types of technology smoothly and efficiently, and this is also the main theme of the present paper.

Lightweight Description Logics for OBDA. One severe obstacle to efficient OBDA is the already mentioned high computational complexity of of query answering with description logics. The situation clearly worsens when the A-Boxes of enterprise ontologies are very large databases. To tackle this problem, new, lightweight DLs have been designed, that guarantee polynomial-time data complexity for conjunctive query answering. This means that based on a fixed T-Box, a fixed query can be answered in polynomial time over variable databases. The best-known and best-studied examples of such lightweight DLs are the DL-Lite family of description logics [4] and \( EL \) (see, e.g., [5]) families. These languages can be considered tractable subclasses of OWL. It was convincingly argued that simple DLs such as DL-Lite or \( EL \) are sufficient for modelling an overwhelming number of applications.

More recently, the Datalog\(^\pm\) family of description logics was introduced [6], [7], [8], [9]. Its syntax is based on classical first-order logic, more specifically, on variants of the well-known Datalog language [10]. The basic Datalog\(^\pm\) rules are known as tuple-generating dependencies (TGDs) in the database literature [11]. Tractable DLs in this framework are guarded Datalog\(^\pm\), which is noticeably more general than both DL-Lite and \( EL \), and the DLs linear Datalog\(^\pm\) and sticky-join Datalog\(^\pm\), which both encompass DL-Lite.

Besides being more expressive than DL-Lite, suitable Datalog\(^\pm\) languages offer a more compact representation of the attributes of concepts and roles, since description logics are usually restricted to unary and binary predicates only. Consider, as an example, a relation \( stock(id, name, unit-price) \). Representing this relation in DL would require the introduction of a concept symbol \( stock \), and of three attribute symbols \( id \), \( name \) and \( unit-price \). These entities must be then woven together by the T-Box formula \( stock \sqsubseteq \exists id \cap \exists name \cap \exists unit-price \). Datalog\(^\pm\) represents the relation in a natural way by means of a ternary predicate \( stock \). In the same way, Datalog\(^\pm\) provides a more natural syntax for many other DL formulæ; for example, an inverse role assertion \( \exists q \sqsubseteq s \) is represented as a full TGD \( r(X,Y) \rightarrow s(Y,X) \), while an existential restriction \( p \sqsubseteq \exists r, q \) is represented as a partial TGD \( p(X) \rightarrow \exists Y r(X,Y), q(Y) \).

First-Order Rewritability. Polynomial-time tractability is often considered not to be good enough for efficient query processing. Ideally, one would like to achieve the same complexity as for processing SQL queries, or, equivalently, first-order (FO) queries. Let\( D \) be a database (i.e., an A-Box), \( \Sigma \) a T-Box based on \( D \), and \( q \) an ontological query on \( \Sigma \). We say that \( \Sigma \) and \( q \) are first-order rewritable (FO-rewritable) if \( \langle \Sigma, q \rangle \) can be compiled into a first-order query \( q_\Sigma \) (called the perfect rewriting) such that \( q_\Sigma \) evaluated over \( D \) yields exactly the same result as \( q \) evaluated against \( \Sigma \) and \( D \). More formally, for every database \( D, D \cup \Sigma \models q \) iff \( D \models q_\Sigma \). Since answering first-order queries is in the class \( AC_0 \) in data complexity [12], it immediately follows that under FO-rewritable TGDs, query answering is also in \( AC_0 \) in data complexity.

This notion was first introduced by Calvanese et al. [4] in the concept of description logics. If a DL guarantees the FO-rewritability of each query under every T-Box, we simply say that the logic is FO-rewritable. FO-rewritability is a most desirable property since it ensures that the reasoning process can be largely decoupled from data access. In fact, to answer query \( q \), a separate software can compile \( q \) into \( q_\Sigma \), and then just submit \( q_\Sigma \) as a standard SQL query to the DBMS holding \( D \), where it is evaluated and optimized in the usual way.

Excitingly, it was shown that the members of the DL-Lite family, as well as the slightly more expressive language linear Datalog\(^\pm\) are FO-rewritable. Moreover, even the much more expressive language of sticky-join Datalog\(^\pm\) is FO-rewritable. For these languages, a pair \( \langle \Sigma, q \rangle \), where \( q \) is a CQ, is rewritten as an SQL expression equivalent to a UCQ \( q_\Sigma \). The research challenge we address in this paper is precisely the question of how to rewrite \( \langle \Sigma, q \rangle \) to \( q_\Sigma \) correctly and efficiently. Let us illustrate this process by a small, but comprehensive example.

A Comprehensive Example: Stock Exchange. Consider the following relational schema \( R \) representing financial information about companies and their stocks:

- \( stock(id, name, unit-price) \)
- \( company(name, country, segment) \)
- \( list_comp(stock, list) \)
- \( fin_idx(name, type, ref-mkt) \)
- \( stock_portf(company, stock, qty) \)

The \( stock \) relation contains information about stocks such as the name, and the price per unit. The relation \( company \) contains information about companies; in particular, the name, the country, and the market segment of a company. The relation \( list_comp \) relates a stock to a financial index (i.e., NASDAQ, FTSE, NIKKEI) represented by the relation \( fin_idx \) which, in turn, contains information about the types of stocks in the index, and the reference market (e.g., London Stock Exchange). Finally, \( stock_portf \) relates companies to their stocks along with an indication of the amount of the investment.

Datalog\(^\pm\) provides the necessary expressive power to extend \( R \) with ontological constraints in an easy and intuitive way. Examples of such constraints follow:

\[
\begin{align*}
\sigma_1 & : stock_portf(X, Y, Z) \rightarrow \exists V \exists W \ company(X, V, W) \\
\sigma_2 & : stock_portf(X, Y, Z) \rightarrow \exists V \exists W \ stock(X, Y, W) \\
\sigma_3 & : list_comp(X, Y) \rightarrow \exists Z \exists W \ fin_idx(Y, Z, W) \\
\sigma_4 & : list_comp(X, Y) \rightarrow \exists Z \exists W \ stock(X, Z, W) \\
\sigma_5 & : stock_portf(X, Y, Z) \rightarrow has_stock(Y, X) \\
\sigma_6 & : has_stock(X, Y) \rightarrow \exists Z stock_portf(Y, X, Z) \\
\sigma_7 & : stock(X, Y, Z) \rightarrow \exists V \exists W \ stock_portf(V, X, W) \\
\sigma_8 & : stock(X, Y, Z) \rightarrow fin_ins(X) \\
\sigma_9 & : company(X, Y, Z) \rightarrow legal_person(X) \\
\delta_1 & : legal_person(X, Y, Z), fin_ins(X, V, W) \rightarrow \bot.
\end{align*}
\]

The first four TGDs set the “domain” and the “range” of the \( stock_portf \) and \( list_comp \) relations, respectively. TGDs
σ₅ and σ₆ assert that stock_portf and has_stock are “inverse relations”, while σ₇ expresses that each stock must belong to some stock portfolio. TGDs σ₈ and σ₉ model taxonomic relationships such as the facts that each stock is a financial instrument, and each company is a legal person. Finally, the negative constraint δ₁ (where ⊥ denotes the truth constant false) states that legal persons and financial instruments are disjoint sets.

Consider now the following conjunctive query q asking for all the triples (a, b, c), where a is a financial instrument owned by the company b and listed on c:

\[ q(A, B, C) \leftarrow \text{fin_ins}(A), \text{stock_portf}(B, A, D), \text{company}(B, E, F), \text{list_comp}(A, C), \text{fin_idx}(C, G, H) \]

Since \( \Sigma = \{\sigma₁, \ldots, \sigma₉\} \) is a set of linear TGDs, i.e., TGDs with single body-atom, query answering under \( \Sigma \) is FO-rewritable. Thus, it is possible to reformulate \( (\Sigma, q) \) to a first-order query \( q₂ \) such that, for every database \( D \), \( D \models \Sigma \models q \) iff \( D \models q₂ \). A naive rewriting procedure would use the TGDs of \( \Sigma \) as rewriting rules for the atoms in \( q \) to generate all the CQs of the perfect rewriting. Table I shows a (partial) rewriting for \( \text{stock}_\text{portf} \) since, due to the existence of the TGDs \( \sigma₂ \) and \( \sigma₈ \) in \( \Sigma \), if the atom \( \text{stock}_\text{portf}(B, A, D) \) is satisfied, then immediately the atom \( \text{fin_ins}(A) \) is also satisfied. Notice that by eliminating a redundant atom from a query, we also eliminate all the atoms in the generated queries, and thus reduce the number of the CQs in the rewritten query. For example, in the given query \( q \) above it is possible to eliminate the atom \( \text{fin_ins}(A) \) since, due to the existence of the TGDs \( \sigma₂ \) and \( \sigma₈ \) in \( \Sigma \), if the atom \( \text{stock}_\text{portf}(B, A, D) \) is satisfied, then immediately the atom \( \text{fin_ins}(A) \) is also satisfied. Notice that by eliminating a redundant atom from a query, we also eliminate all the atoms that are generated starting from it during the rewriting process. Moreover, due to the TGD \( \sigma₃ \), if the atom \( \text{list}_\text{comp}(A, C) \) in \( q \) is satisfied, then the atom \( \text{fin_idx}(C, G, H) \) is also satisfied, and therefore can be eliminated. Finally, due to the TGD \( \sigma₁ \), if the atom \( \text{stock}_\text{portf}(B, A, D) \) is satisfied, then the atom \( \text{company}(B, E, F) \) is also satisfied, and hence is redundant. The query that has to be considered as input of the rewriting process is therefore \( q(A, B, C) \leftarrow \text{stock_portf}(B, A, D), \text{list_comp}(A, C) \) that produces a perfect rewriting containing the following two queries executing only two joins:

\[ q(A, B, C) \leftarrow \text{list_comp}(A, C), \text{stock_portf}(B, A, D) \]
\[ q(A, B, C) \leftarrow \text{list_comp}(A, C), \text{has_stock}(A, B) \]

**Contributions and Roadmap.** After a review of previous work on ontology-based data access in the next section, and some formal definitions and preliminaries in Section III, we present a short overview of the Datalog family in Section IV. We then proceed with new research results. In Section V, we propose a new rewriting algorithm that improves the one stated in [13] by substantially reducing the number of redundant queries in the perfect rewriting. In Section VI, we present a polynomial-time optimization strategy based on the early-pruning of redundant atoms produced during the rewriting process. An implementation and experimental evaluation of the new method is discussed in Section VII. We also discuss the relationship between our optimization technique and optimal query minimization algorithms such as the chase & back-chase algorithm [14]. We conclude with a brief outlook on further research.

**II. Ontology Based Data Access**

Answering queries under constraints and the related optimization techniques are important topics in data management beyond the obvious research interest. In particular, they are profitable opportunities for companies that need to deliver efficient and effective data management solutions to their customers. This trend is becoming even more evident as a plethora of robust systems and APIs for Semantic Web data management proposed in the recent years. These systems span from open-source solutions such as Virtuoso², Sesame³, RDFSuite [15], KAON⁴ and Jena⁵, to commercial implementations such as the semantic extensions implemented in Oracle Database 11g R2 [16] and BigOWLIm⁶. In this Section we briefly analyze the systems providing rewriting-based access to databases under ontological constraints, and we highlight some crucial points that we want to address in this work.

We first present the class of constraints identified by the members of the DL-Lite family [4], namely, DL-Lite₄, DL-Liteᵣ, and DL-Liteᵣ₉, underlying the W3C OWL-QL profile of

²http://virtuoso.openlinksw.com/
³http://www.openrdf.org/
⁴http://kaon.semanticweb.org/.
⁵http://jena.sourceforge.net/
⁶http://www.ontotext.com/owlim/
the OWL language. These constraints correspond to unary and binary inclusion dependencies combined with a restricted for of key constraints. In order to perform query answering under this class of constraints, a rewriting algorithm, introduced in [4] and implemented in the QuOnto system, reformulates the given query into unions of conjunctive queries. The size of the reformulated query is unnecessarily large due to a number of reasons. In the first place, (i) basic optimization techniques such as the identification of the connected components in the body of the input query, or the computation of any form of query decomposition [17], are not applied. Moreover, (ii) the fact that the given set of constraints can be used to identify existential joins in the reformulated query which can be eliminated is not exploited. Finally, (iii) the factorization step (which is needed to guarantee completeness) is applied exhaustively, and as a result many superfluous queries are generated.

Peréz-Urbina et al. [18] proposed an alternative resolution-based rewriting algorithm, implemented in the Requiem system, that addressed 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. The algorithm has then been extended to more expressive DL languages [18]. In this case the output of the rewriting is a Datalog program.

Rosati et al. [19] recently proposed a very sophisticated rewriting technique, implemented in the Presto system, that addresses some of the issues described above. In particular, (i) the unnecessary existential joins are eliminated by resorting to the concept of most-general subsumees, which also avoids the unnecessary factorizations, and (ii) the connectivity of the given query is checked before executing the algorithm; in case the query is not connected, Presto splits the query in connected components and rewrites them separately. Notice that Presto produces a non-recursive Datalog program, and not a union of conjunctive queries. This allows the “hiding” of the exponential blow-up inside the rules instead of generating explicitly the disjunctive normal form. The final rewriting is exponential only in the number of non-eliminable existential joins, but not in the size of the input query.

The approaches presented above have been proven very effective when applied to very particular classes of description logic constraints. Following a more general approach for ontological query answering, Cali et al. [13] presented a backward-chaining rewriting algorithm which is able to deal with arbitrary sets of TGDs, providing that the class of TGDs under consideration satisfies suitable syntactic restrictions that guarantee the termination of the algorithm. However, this algorithm is inspired by the original QuOnto algorithm and inherits all its drawbacks.

Despite the possibly exponential number of queries to be constructed, we know that all such queries are independent from each other, and thus they can be easily executed in parallel threads and distributed on multiple processors. Notice that a non-recursive Datalog program is not equally easy to distribute. Moreover, the optimizations implemented in current DBMS systems for (unions of) conjunctive queries are much more advanced than those implemented for the positive existential first-order queries resulting from the translation of a non-recursive Datalog program into a concrete query language such as SQL. It is clear that a trade-off between these two approaches must be found in order to exploit as much as possible the current optimization techniques, while keeping the size of the rewriting reasonably small in order to make the execution of it feasible in practice.

A related research field is that of query minimization [20], in particular, in presence of views and constraints [21], [14]. Given a conjunctive query $q$, and a set of constraints $\Sigma$, the goal is to find all the minimal equivalent reformulations of $q$ under the constraints of $\Sigma$. The most interesting approach in this respect is the chase & back-chase algorithm (C&B) [14], implemented in the MARS system [22]. The algorithm freezes the atoms of body($q$) and, by considering them as a database $D_q$, applies the following two steps. During the chase-step, the chase of $D_q$ w.r.t. $\Sigma$ is constructed, and then the atoms of chase($D_q, \Sigma$) are considered as the body-atoms of a query $q_u$, called the universal plan. The back-chase step considers all the possible subsets of the atoms of body($q_u$), starting from those with a single-atom, which are then considered as the body of a query $q'$. Whenever there exists a containment mapping from body($q_u$) to chase($D_{q'}$, $\Sigma$), where $D_{q'}$ is the database obtained by freezing body($q'$), then $q'$ is an equivalent reformulation of $q$. Moreover, every time an equivalent reformulation $q'$ is found, the back-chase does not consider any of the supersets of the atoms of body($q'$) because they will be automatically implied by the atoms of $q'$, and therefore the produced query would be redundant. This particular exploration strategy guarantees the minimality of the reformulations. A non-negligible drawback of this approach is the fact that we need to compute the chase of $D_q$ w.r.t. $\Sigma$, and also the chase for the (exponentially many) databases $D_{q'}$ w.r.t. $\Sigma$. Clearly, this makes the procedure computationally expensive.

III. Preliminaries

In this section we recall some basics on relational databases, conjunctive queries, tuple-generating dependencies, and the chase procedure.

A. Relational Databases and Conjunctive Queries

Consider two pairwise disjoint (infinite) sets of symbols $\Delta_c$ and $\Delta_z$ such that: $\Delta_c$ is a set of constants (which constitutes the domain of a database), and $\Delta_z$ is a set of labeled nulls (used as placeholders for unknown values). Different constants represent different values (unique name assumption), while different nulls may represent the same value. Throughout the paper, we denote by $X$ sequences of variables $X_1, \ldots, X_k$, where $k \geq 0$, and by $[n]$ the set $\{1, \ldots, n\}$, for any $n \geq 1$.

A relational schema $R$ (or simply schema) is a set of relational symbols (or predicate symbols), each with its associated arity. A position $r[i]$ (or $\langle r, i \rangle$) is identified by a predicate $r \in R$ and its $i$-th argument. A term $t$ is a constant, labeled
null, or variable. An atomic formula (or simply atom) has the form $r(t_1,\ldots,t_n)$, where $r \in R$ has arity $n$, and $t_1,\ldots,t_n$ are terms. Conjunctions of atoms are often identified with the sets of their atoms.

A substitution from one set of symbols $S_1$ to another set of symbols $S_2$ is a function $h : S_1 \to S_2$. A homomorphism from a set of atoms $A_1$ to a set of atoms $A_2$, both over the same schema $\mathcal{R}$, is a substitution $h$ from the set of terms of $A_1$ to the set of terms of $A_2$ such that: (i) if $t \in \Delta_r$, then $h(t) = t$, and (ii) if $r(t_1,\ldots,t_n)$ is in $A_1$, then $h(r(t_1,\ldots,t_n)) = r(h(t_1),\ldots,h(t_n))$ is in $A_2$. The notion of homomorphism naturally extends to conjunctions of atoms.

A relational instance (or simply instance) $I$ for a schema $\mathcal{R}$ is a (possibly infinite) set of atoms of the form $r(t)$, where $r \in \mathcal{R}$ has arity $n$ and $t \in (\Delta_r \cup \Delta_s)^n$. A database is a finite relational instance. A conjunctive query (CQ) $q$ of arity $n$ over a schema $\mathcal{R}$ is a formula of the form $q(X) \leftarrow \phi(X,Y)$, where $\phi(X,Y)$ is a conjunction of atoms over $\mathcal{R}$, and $q$ is an $n$-ary predicate. $\phi(X,Y)$ is called the body of $q$, denoted as $body(q)$, and $q(X)$ is the head of $q$, denoted as $head(q)$. A Boolean conjunctive query (BCQ) is a CQ of arity zero. The answer to a CQ $q$ of arity $n$ over an instance $I$, denoted as $q(I)$, is the set of all $n$-tuples $t \in (\Delta_r)^n$ for which there exists a homomorphism $h : X \cup Y \to \Delta_r \cup \Delta_s$ such that $h(\phi(X,Y)) \subseteq I$ and $h(X) = t$. A BCQ has only the empty tuple $\emptyset$ as possible answer, in which case it is said that $q$ satisfies the answer relationally. Formally, a BCQ has positive answer over $I$, denoted as $I \models q$, iff $\emptyset \in q(I)$. A union of CQs (UCQ) $Q$ of arity $n$ is a set of CQs, where each $q \in Q$ has the same arity $n$ and uses the same predicate symbol in the head. The answer to $Q$ over an instance $I$, denoted as $Q(I)$, is defined as the set of tuples $\{ t \mid \text{there exists } q \in Q \text{ such that } t \in q(I) \}$.

B. Tuple-Generating Dependencies

A tuple-generating dependency (TGD) $\sigma$ over a schema $\mathcal{R}$ is a first-order formula $\forall X \forall Y \phi(X,Y) \to \exists Z \psi(X,Z)$, where $\phi(X,Y)$ and $\psi(X,Z)$ are conjunctions of atoms over $\mathcal{R}$, called the body and the head of $\sigma$, denoted as $body(\sigma)$ and $head(\sigma)$, respectively. Henceforth, to avoid notational clutter, we will omit the universal quantifiers in TGDs. Such $\sigma$ is satisfied by an instance $I$ for $\mathcal{R}$ iff, whenever there exists a homomorphism $h$ such that $h(\phi(X,Y)) \subseteq I$, there exists an extension $h'$ of $h$ (i.e., $h' \supseteq h$) such that $h'(\psi(X,Z)) \subseteq I$.

We now define the notion of query answering under TGDs. Given a database $D$ for $\mathcal{R}$, and a set $\Sigma$ of TGDs over $\mathcal{R}$, the models of $D$ w.r.t. $\Sigma$, denoted as $mods(D,\Sigma)$, is the set of all instances $I$ such that $I \models D \cup \Sigma$, which means that $I \supseteq D$ and $I$ satisfies $\Sigma$. The answer to a CQ $q$ w.r.t. $D$ and $\Sigma$, denoted as $ans(q,D,\Sigma)$, is the set $\{ t \mid t \in q(I) \text{ for each } I \in mods(D,\Sigma) \}$. The answer to a BCQ $q$ w.r.t. $D$ and $\Sigma$ is positive, denoted as $D \cup \Sigma \models q$, iff $ans(q,D,\Sigma) \neq \emptyset$. Note that query answering under general TGDs is undecidable [23], even when the schema and the set of TGDs are fixed [24]. We recall that the two problems of answering CQs and BCQs under TGDs are equivalent [20], [25]. Roughly speaking, we can enumerate the polynomially many tuples of constants which are possible answers to $q$, and then, instead of answering the given query $q$, we answer the polynomially many BCQs that we obtain by replacing the variables in the body of $q$ with the appropriate constants. A certain tuple $t$ of constants is in the answer of $q$ iff the answer to the BCQ that we obtain from $t$ is positive. Henceforth, we thus focus only on the BCQ answering problem.

C. The TGD Chase

The chase procedure (or simply chase) is a fundamental algorithmic tool introduced for checking implication of dependencies [26], and later for checking query containment [27]. Informally, the chase is a process of repairing a database w.r.t. a set of dependencies so that the resulted database satisfies the dependencies. 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 a database $D$ for a schema $\mathcal{R}$, and a TGD $\sigma = \phi(X,Y) \rightarrow \exists Z \psi(X,Z)$ over $\mathcal{R}$. If $\sigma$ is applicable to $D$, i.e., there exists a homomorphism $h$ such that $h(\phi(X,Y)) \subseteq D$ then: (i) define $h^2 = h$ such that $h^2(z_i) = z_i$, for each $z_i \in Z$, where $z_i \in \Delta_r$, is a “fresh” labeled null not introduced before, and (ii) add to $D$ the set of atoms in $h^2(\psi(X,Z))$, if not already in $D$.

Given a database $D$ and a set of TGDs $\Sigma$, the chase algorithm for $D$ and $\Sigma$ consists of an exhaustive application of the TGD chase rule in a breadth-first fashion, which leads as result to a (possibly infinite) chase for $D$ and $\Sigma$, denoted as chase($D,\Sigma$). For the formal definition of the chase algorithm we refer the reader to [7].

The (possibly infinite) chase for $D$ and $\Sigma$ is a universal model of $D$ w.r.t. $\Sigma$, i.e., for each instance $I \in mods(D,\Sigma)$, there exists a homomorphism from chase($D,\Sigma$) to $I$ [28], [25]. Using this fact it can be shown that $D \cup \Sigma \models q$ iff chase($D,\Sigma$) $\models q$, for every BCQ $q$.

IV. THE DATALOG$^\pm$ FAMILY

In this section we present the main Datalog$^\pm$ languages under which query answering is decidable, and (almost in all cases) also tractable in data complexity.

A. Decidability Paradigms

We first discuss the three main paradigms for ensuring decidability of query answering, namely, chase termination, guardedness and stickiness.

Chase Termination. In this case the chase always terminates and produces a finite universal model $U$. Thus, given a query we just need to evaluate it over the finite database $U$. The most notable syntactic restriction of TGDs guaranteeing chase termination is weak-acyclicity, which is defined by means of a graph-based condition, for which we refer the reader to the landmark paper [28]. Roughly speaking, in the chase constructed under a weakly-acyclic set of TGDs over a schema $\mathcal{R}$, only a finite number of distinct values can appear at any position of $\mathcal{R}$, and thus after finitely many steps the chase procedure terminates. It is known that query answering
under a weakly-acyclic set of TGDs is \( \text{PTIME}\)-complete [28] and \( \text{2EXPTIME}\)-complete [9] in data and combined complexity, respectively. More general syntactic restrictions that guarantee chase termination were proposed in [25] and [29].

**Guardedness.** Guarded TGDs, introduced and studied in [24], have an atom in their body, called the **guard**, that contains all the universally quantified variables. For example, the TGD \( r(X, Y), s(X, Y, Z) \rightarrow \exists W s(Z, X, W) \) is guarded via the guard atom \( s(X, Y, Z) \), while the TGD \( r(X, Y), r(Y, Z) \rightarrow r(X, Z) \) is not. Decidability of query answering follows from the fact that the chase constructed under a set of guarded TGDs has the bounded treewidth property, i.e., is a “tree-like” structure. The data and combined complexity of query answering under a set of guarded TGDs is \( \text{PTIME}\)-complete [6] and \( \text{2EXPTIME}\)-complete [24], respectively.

**Linear TGDs.** Proposed in [6], is a FO-rewritable variant of guarded TGDs. A TGD is linear iff it contains only one atom in its body. Obviously a linear TGD is trivially guarded since the singleton body-atom is automatically a guard. Linear TGDs are more expressive than the well-known class of inclusion dependencies. Query answering under linear TGDs is in the highly tractable class \( \text{AC}_0 \) in data complexity [6]. The same problem is \( \text{PSPACE}\)-complete in combined complexity; this result is immediately implied by results in [27].

An expressive class, which is a generalization of guarded TGDs, is the class of **weakly-guarded** sets of TGDs introduced in [24]. Intuitively speaking, a set \( \Sigma \) of TGDs is weakly-guarded iff in the body of each TGD of \( \Sigma \) there exists an atom, called the **weak-guard**, that contains all the universally quantified variables that appear only at positions where a “fresh” null of \( \Delta \) can appear during the construction of the chase. Query answering under a weakly-guarded set of TGDs is \( \text{EXPTIME}\)-complete [24] and \( \text{2EXPTIME}\)-complete [24] in data and combined complexity, respectively.

**Stickiness.** In this paragraph we present a Datalog\(^\pm\) language (and its extensions), which hinges on a paradigm that is very different from guardedness. **Sticky** sets of TGDs are defined formally by an efficiently testable condition involving variable-marking [8]. In what follows we just give an intuitive definition of this class. For every database \( D \), assume that during the construction of chase of \( D \) under a sticky set of TGDs, we apply a TGD \( \sigma \in \Sigma \) that has a variable \( V \) appearing more than once in its body; assume also that \( V \) maps (via homomorphism) on the symbol \( z \), and that by virtue of this application the atom \( \alpha \) is introduced. In this case, for each atom \( \beta \) in body(\( \sigma \)), we say that \( \alpha \) is derived from \( \beta \). Then, we have that \( z \) appears in \( \alpha \) and in all atoms resulting from some chase derivation sequence starting from \( \alpha \), “sticking” to them (hence the name “sticky” sets of TGDs). Interestingly, sticky sets of TGDs are FO-rewritable, and thus query answering is feasible in \( \text{AC}_0 \) in data complexity [8]. Combined complexity of query answering is known to be \( \text{EXPTIME}\)-complete [8].

In [9] the FO-rewritable class of **sticky-join** sets of TGDs, that captures both linear TGDs and sticky sets of TGDs, is introduced. Similarly to sticky sets of TGDs, sticky-join sets are defined formally by a testable condition based on variable-marking. However, this variable-marking procedure is more sophisticated than the one used for sticky sets, and due to this fact the problem of identifying sticky-join sets of TGDs is harder than the one of identifying sticky sets. In particular, given a set \( \Sigma \) of TGDs, we can decide in \( \text{PTIME} \) whether \( \Sigma \) is sticky, while the problem whether \( \Sigma \) is sticky-join is \( \text{2EXPTIME}\)-complete. Note that the data and combined complexity of query answering under sticky and sticky-join sets of TGDs coincide.

**B. Additional Features**

In this subsection we briefly discuss how the languages presented above can be combined with negative constraints and key dependencies, without altering the complexity of query answering.

**Negative Constraints.** A negative constraint (NC) \( \nu \) over a schema \( \mathcal{R} \) is a first-order formula \( \forall X \phi(X) \rightarrow \bot \), where \( \bot \) denotes the truth constant false. NCs are vital when representing ontologies (see, e.g., [6], [8]), as well as conceptual schemas such as Entity-Relationship diagrams (see, e.g., [30], [31]). With NCs we can assert, for example, that students and professors are disjoint sets: \( \forall X \text{ student}(X), \text{ professor}(X) \rightarrow \bot \). Also, we can state that a student cannot be the leader of a research group: \( \forall X \forall Y \text{ student}(X), \text{ leader}(X, Y) \rightarrow \bot \).

It is shown that checking NCs is tantamount to query answering [6]. In particular, given an instance \( I \), a set \( \Sigma_\perp \) of NCs, and a set \( \Sigma \) of TGDs, for each NC \( \nu \) of the form \( \forall X \phi(X) \rightarrow \bot \), we answer the BCQ \( q_\nu() \leftarrow \phi(X) \). If at least one of such queries answers positively, then \( I \cup \Sigma \cup \Sigma_\perp \models \bot \) (i.e., the theory is inconsistent), and therefore \( I \cup \Sigma \cup \Sigma_\perp \models q \) for every BCQ \( q \); otherwise, given a BCQ \( q \), we have \( I \cup \Sigma \cup \Sigma_\perp \models q \) iff \( I \cup \Sigma \models q \), i.e., we can answer \( q \) by ignoring the set of NCs.

**Key Dependencies.** It is well-known that the interaction of general TGDs and key dependencies (KDs) leads to undecidability of query answering [32]; we assume that the reader is familiar with the notion of KD (see, e.g., [33]). Thus, the classes of TGDs presented above cannot be combined arbitrarily with KDs. Suitable syntactic restrictions are needed in order to ensure decidability of query answering.

A crucial concept towards this direction is separability [34], which formulates a controlled interaction of TGDs and KDs. Formally speaking, a set \( \Sigma = \Sigma_T \cup \Sigma_K \) over a schema \( \mathcal{R} \), where \( \Sigma_T \) and \( \Sigma_K \) are sets of TGDs and KDs, respectively, is separable iff for every instance \( I \) for \( \mathcal{R} \), either \( I \) violates \( \Sigma_K \), or for every BCQ \( q \) over \( \mathcal{R} \), \( I \cup \Sigma \models q \) iff \( I \cup \Sigma_T \models q \). Notice that separability is a semantic notion. A sufficient syntactic criterion for separability of TGDs and KDs is given in [6]: TGDs and KDs satisfying the criterion are called non-conflicting.

Obviously, in case of non-conflicting sets of TGDs and KDs, we just need to perform a preliminary check whether the given instance satisfies the KDs, and if this is the case, then we eliminate them, and proceed by considering only the set of TGDs. This preliminary check can be done using negative constraints. For example, to check whether the KD \( \text{key}(r) = \{1\} \), stating
that the first attribute of the binary relation \( r \) is a key attribute, is satisfied by the database \( D \), we just need to check whether the database \( D \) obtained by adding to \( D \) the set of atoms \( \{ \text{neq}(a, b) \mid a \neq b \) and \( a, b \) are constants occurring in \( D \)\), where \( \text{neq} \) is an auxiliary predicate, satisfies the negative constraint \( r(X, Y), r(X, Z), \text{neq}(Y, Z) \rightarrow \bot \). The atom \( \text{neq}(a, b) \) implies that \( a \) and \( b \) are different constants. Since, as already mentioned, checking NCs is tantamount to query answering, we immediately get that the complexity of query answering under non-conflicting sets of TGDs and KDs is the same as in the case of TGDs only.

Interestingly, by combining non-conflicting linear (or sticky) sets of TGDs and KDs with NCs, we get strictly more expressive formalisms than the most widely-adopted tractable ontology languages, in particular DL-Lite\(_A\), DL-Lite\(_X\) and DL-Lite\(_{\mathcal{R}}\), without loosing FO-rewritability, and consequently highly tractability of query answering in data complexity. For more details, we refer the interested reader to [6], [8].

V. DATALOG\( ^\pm \) FOR OBDA

We consider the problem of BCQ answering under the FO-rewritable members of the Datalog\( ^\pm \) family, namely, linear, sticky and sticky-join sets of TGDs. Given a BCQ \( q \) and a set of TGDs \( \Sigma \), the actual computation of the rewriting is done by applying a backward-chaining resolution procedure using the rules of \( \Sigma \) as rewriting rules. Our algorithm optimizes the algorithm presented in [13] by greatly reducing the number of BCQs in the rewriting, and therefore improves the overall performance of query answering. Before going into the details of the rewriting algorithm we first give some useful notions.

Given two atoms \( a, b \) we say that they unify if there exists a substitution \( \gamma \), called unifier for \( a \) and \( b \), such that \( \gamma(a) = \gamma(b) \).

A most general unifier (MGU) is a unifier for \( a \) and \( b \), denoted as \( \gamma_{a,b} \), such that for each other unifier \( \gamma \) for \( a \) and \( b \), there exists a substitution \( \gamma' \) such that \( \gamma = \gamma' \circ \gamma_{a,b} \). Notice that if two atoms unify, then there exists a MGU. Furthermore, the MGU for two atoms is unique up to variable renaming.

BCQ answering under (general) TGDs and TGDs with just one atom in their heads are equivalent problems [24]. This is established by providing a LOGSPACE transformation from (general) TGDs to TGDs with singleton head-atoms. Since this transformation preserves the syntactic condition of linear, sticky and sticky-join sets of TGDs, henceforth we assume w.l.o.g. that every TGD has just one atom in its head. We now give the notion of applicability of a TGD to an atom of a query. Let us assume w.l.o.g. that the variables that appear in the query, and those that appear in the TGD, constitute two disjoint sets. Given a BCQ \( q \), a variable is called shared in \( q \) if it occurs more than once in \( \text{body}(q) \).

**Definition 1 (Applicability [13]):**

Let \( \sigma \) be a TGD over a schema \( \mathcal{R} \), and \( q \) a BCQ over \( \mathcal{R} \). Given an atom \( a \in \text{body}(q) \), we say that \( \sigma \) is applicable to \( a \) whenever \( a \) and \( \text{head}(\sigma) \) unify, where \( \gamma_{a,\sigma} \) is the MGU for \( a \) and \( \text{head}(\sigma) \), and the following conditions are satisfied:

1. If the term at position \( \pi \) is either a constant, or a shared variable in \( \text{body}(q) \) that occurs also in some atom of \( \text{body}(q) \) other than \( a \), then the variable at position \( \pi \) in \( \text{head}(\sigma) \) occurs in \( \text{body}(\sigma) \).

2. If a shared variable in \( q \) occurs only in \( a \) at positions \( \pi_1, \ldots, \pi_m \), for \( m \geq 2 \), then either, for each \( i \in [m] \), the variable at position \( \pi_i \) in \( \text{head}(\sigma) \) occurs also in \( \text{body}(\sigma) \), or at positions \( \pi_1, \ldots, \pi_m \) in \( \text{head}(\sigma) \) we have the same existentially quantified variable.

Let us now introduce the notion of factorizability which, as we explain below, makes one of the main differences between our algorithm and the one presented in [13], due to which a perfect rewriting with less BCQs is obtained.

**Definition 2 (Factorizability):**

Given a query \( q \) and a set of TGD \( \Sigma \), we say that a position \( \pi \) in an atom \( a \in \text{body}(q) \) is existential if there exists a TGD \( \sigma \) such that \( a \) unifies with \( \text{head}(\sigma) \), and the term at position \( \pi \) in \( \text{head}(\sigma) \) is an existentially quantified variable. A set of atoms \( S \subseteq \text{body}(q) \) is factorizable w.r.t a TGD \( \sigma \) iff the following conditions hold:

1. For each pair of atoms \( a, b \) of \( S \), \( a \) and \( b \) unify.

2. If a variable \( V \) appears at an existential position in an atom of \( S \), then \( V \) does not occur in \( \text{body}(q) \setminus S \), and also \( V \) occurs only at existential positions.

**Example 1 (Factorization):**

Consider the three BCQs

\[
\begin{align*}
q_1 & : q() \leftarrow t(A, C, B), t(A, E, C) \\
q_2 & : q() \leftarrow s(C), t(A, C, B), t(A, E, C) \\
q_3 & : q() \leftarrow t(A, C, B), t(C, E, B)
\end{align*}
\]

and the TGD \( \sigma : s(X), r(X, Y) \rightarrow \exists Z t(X, Z, Z) \). Clearly, the atoms in the body of \( q_1 \) unify through the MGU \( \gamma = \{ E \rightarrow C, B \rightarrow C \} \), and they are also factorizable since the variables \( C, B \) and \( E \) appear only at existential positions in \( q_1 \).

The factorization results in the query \( q_1' : q() \leftarrow t(A, C, C) \); notice that \( \sigma \) is not applicable to \( q_1' \) but it is applicable to \( q_1 \). On the contrary, despite the fact that the atoms \( t(A, C, B) \) and \( t(A, E, C) \) in \( q_2 \) unify, since the variable \( C \) appears also in position \( s[1] \) which is not existential w.r.t \( \sigma \), the atoms are not factorizable. The same holds for \( q_3 \) where the atoms \( t(A, C, B) \) and \( t(C, E, B) \) unify but the variable \( C \) appears in \( t[1] \) which is not existential w.r.t \( \sigma \).

We are now ready to describe the algorithm TGD-\text{rewrite} whose pseudo-code is presented as Algorithm 1, and is based, besides factorization, on the algorithm of [13]. The perfect rewriting of a BCQ \( q \) is computed by exhaustively expanding the atoms in \( \text{body}(q) \) by means of the applicable TGDs in \( \Sigma \). Each application of a TGD leads to a new BCQ which is implied (w.r.t \( \Sigma \)) by the given query \( q \); these queries are stored into a set \( Q_r \). The process is repeated until no new queries can be generated (up to variable renaming). The union of
Algorithm 1 The algorithm TGD-rewrite

Require: a schema $\mathcal{R}$, a set of TGDs $\Sigma$ and a BCQ $q$ over $\mathcal{R}$
Ensure: the FO-rewriting $Q_r$ of $q$ under $\Sigma$

1. $Q_{new} := \{q\}$
2. $Q_r := \emptyset$

repeat
3. $Q_c := Q_r \cup Q_{new}$
4. $Q_{temp} := Q_{new}$
5. $Q_{new} := \emptyset$
6. for all $q \in Q_{temp}$ do
7. for all $\sigma \in \Sigma$ do
8. if $\sigma$ is applicable to $q$ then
9. $q' := \text{factorize}(q, \sigma)$
10. $Q_n := Q_n \cup \{q''\}$
11. end if
12. end for
13. until $Q_{new} = \emptyset$
14. return $Q_r$

BCQs $Q_r$ is the perfect rewriting of $q$ w.r.t. $\Sigma$. The algorithm exhaustively applies two steps: factorization and rewriting.

**Factorization Step.** Each new query $q$ is checked for factorization. Then, we proceed to the rewriting step using the factorized query $q'$. The function $\text{factorize}(q, \sigma)$ identifies the maximal subsets $S_1, \ldots, S_n$ of $\text{body}(q)$, for $n \geq 0$, which are factorizable w.r.t. $\sigma$. If $n > 0$, i.e., there exists at least one such subset, then the query $q'$, obtained by applying the MGU for the atoms of $S_i$ on $q$, where $i \in [n]$ is chosen arbitrarily, is returned. Note that in the case where $n = 0$, i.e., there is no a factorizable subset of $\text{body}(q)$ w.r.t. $\sigma$, then the returned query is $q$.

**Rewriting Step.** The rewriting step is applied after the factorization. During an application of the rewriting step, if there exists a TGD $\sigma$ and a BCQ $q \in Q_{new}$ containing an atom $a$ such that $\sigma$ is applicable to $a$, then the algorithm constructs a new query $q'' = \gamma\tau_{a}(\gamma[q/body(\sigma)])$, that is, the BCQ obtained from $q$ by replacing $a$ with $\text{body}(\sigma)$, and then apply the MGU for $a$ and $\text{head}(\sigma)$. All the generated queries are added to the set of BCQs $Q_{new}$.

**Example 2 (Rewriting):** Consider the set $\Sigma$ of TGDs

- $\sigma_1 : p(X) \rightarrow \exists Y \exists Z t(X,Y,Z)$
- $\sigma_2 : t(X,Y,Z) \rightarrow r(Y,Z)$

and the query $q() \leftarrow t(A,B,C), r(B,C)$. TGD-rewrite first applies $\sigma_2$ to the atom $r(B,C)$ since $\sigma_1$ is not applicable. The query $q_1: q() \leftarrow t(A,B,C), t(V^1, B, C)$ is produced, whose atoms are factorizable through the MGU $\gamma = \{V^1 \rightarrow A\}$. The factorized query is $q_2: q() \leftarrow t(A,B,C)$. Now, $\sigma_1$ is applicable to the body-atom of $q_2$ since the variables $B$ and $C$ at the existential positions are non-shared. The obtained query is $q_3: q() \leftarrow s(A)$. The perfect rewriting constructed by the algorithm is the set $\{q, q_2, q_3\}$.

The next example shows that dropping the applicability condition, then TGD-rewrite may produce unsound rewritings.

**Example 3 (Loss of soundness):** Consider the set $\Sigma$ of TGDs given in Example 2, and also the BCQ $q_1: q() \leftarrow t(A, B, c)$, where $c$ is a constant of $\Delta_c$. Dropping condition (1) would lead to the BCQ $q': q() \leftarrow s(A)$, where the information about the constant $c$ is lost. Consider now a database $D = \{s(b), t(a, b, d)\}$ for $\mathcal{R}$. The query $q'$ maps to the atom $s(b)$ which implies that $D \models q'$. However, the original query $q$ does not map to $\text{chase}(D, \Sigma)$, and thus $D \cup \Sigma \not\models q$. Therefore, any rewriting containing $q'$ is not a sound rewriting of $q$ given $\Sigma$.

Consider now the query $q'' : q() \leftarrow t(A, B, B)$. Dropping condition (2) would lead to the same query $q'$ mapping to the atom $s(b)$ of $D$. However, during the construction of $\text{chase}(D, \Sigma)$ it is not possible to get an atom of the form $t(X, Y, Y)$, where at positions $t[2]$ and $t[3]$ the same value occurs. This implies that there is no homomorphism that maps $q$ to $\text{chase}(D, \Sigma)$, and hence $D \cup \Sigma \not\models q$. Therefore, any rewriting containing $q''$ is again unsound.

The applicability condition may prevent the generation of queries that are vital to guarantee completeness of the rewritten query, as shown by the following example. This is exactly the reason why the factorization step is also needed.

**Example 4 (Loss of completeness):** Consider the set $\Sigma$ of TGDs

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

and the query $q : q() \leftarrow t(A, B), s(B)$. The only viable strategy in this case is to apply $\sigma_2$ to the atom $s(B)$ in $q$, since the atom $t(A, B)$ is blocked by the applicability condition due to the shared variable $B$. The query that we obtain is $q' : q() \leftarrow t(A, B), t(X, B)$, where $X$ is a fresh variable. Notice that in $q'$ the variable $B$ remains shared thus it is not possible to apply $\sigma_1$. It is obvious that without the factorization step there is no way to obtain the query $q'' : q() \leftarrow p(A)$ during the rewriting process. Now, consider the database $D = \{p(a)\}$. Clearly, $\text{chase}(D, \Sigma) = \{p(a), t(a, z_1), s(z_1)\}$, and therefore $\text{chase}(D, \Sigma) \models q$, or, equivalently, $D \cup \Sigma \models q$. However, the rewritten query is not entailed by the given database $D$, since $q''$ does not belong to it, which implies that is not complete.

**Theorem 1:** Let $\mathcal{R}$ be a relational schema. Consider a BCQ $q$ over $\mathcal{R}$, a database $D$ for $\mathcal{R}$, and a set $\Sigma$ of TGDs over $\mathcal{R}$. Then, $D \models \text{TGD-rewrite}(\mathcal{R}, \Sigma, q)$ iff $D \cup \Sigma \models q$. 

proof (sketch). Soundness is proved by induction on the number of applications of the rewriting step. On the other hand, completeness is proved by induction on the number of applications of the chase rule; recall that $D \cup \Sigma \models q$ is equivalent to say that chase$(D, \Sigma) \models q$. \hfill $\Box$

The above result holds for arbitrary TGDs. However, termination is guaranteed if we consider linear, sticky or sticky-join sets of TGDs since, during the rewriting process, only finitely many non-equivalent queries are generated.

**Theorem 2:** The algorithm TGD-rewrite terminates under linear, sticky or sticky-join sets of TGDs.

Approaches such as those of [4] and [13] resort to exhaustive factorizations of the atoms in the queries generated by the rewriting algorithm. By factorizing a query $q$ we obtain a subquery $q'$, that is, $q$ implies $q'$ (w.r.t. the given set of TGDs). Observe that by computing the factorized query $q'$ we eliminate unnecessary shared variables, in the body of $q$, due to which the applicability condition is violated. Consider for example the query $q'$ of Example 4. By factorizing the two atoms with the substitution $X \rightarrow A$ we obtain the query $q'() \leftarrow t(A, B)$ which is a subquery (w.r.t. to the given set of TGDs) of $q'$ (in this case equivalent to $q'$), where the variable $B$ is no longer shared. Thus, the rewriting step can now apply $\sigma_1$ to $t(A, B)$ and produce the query $q'() \leftarrow p(A)$ which is needed to ensure completeness.

The exhaustive factorization produces a non-negligible number of redundant queries as demonstrated by the simple example above. It is thus necessary to apply a restricted form of factorization that generates a possibly small number of BCQs that are necessary to guarantee completeness of the rewritten query. 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 thus unified with no loss of information. The key principle behind our factorization process is that, in order to be applied, there must exist a TGD that can be applied to the output of the factorization.

**VI. Rewriting Optimization**

It is common knowledge that the perfect rewriting obtained by applying a backward-chaining rewriting algorithm (like TGD-rewrite) is, in general, not very well-suited for execution by a DB engine due to the large number of queries to be evaluated. In this section we propose a technique, called query elimination, aiming at optimizing the obtained rewritten query under the class of linear TGDs. As we shall see, query elimination (which is an additional step during the execution of the algorithm TGD-rewrite) reduces (i) the number of BCQs of the perfect rewriting, (ii) the number of atoms in each query of the rewriting as well as (iii) the number of joins. Note that in the rest of the paper we restrict our attention on linear TGDs. Recall that linear TGDs are TGDs with just one atom in their body. Since we also assume, as explained in the previous section, TGDs with just one atom in their head, henceforth, when using the term TGD, we shall refer to TGDs with just one body-atom and one head-atom.

By exploiting the given set of TGDs, it is possible to identify atoms in the body of a certain query that are logically implied (w.r.t. the given set of TGDs) by other atoms in the same query. In particular, for each BCQ $q$ obtained by applying the rewriting step of TGD-rewrite, the atoms of body$(q)$ that are logically implied (w.r.t. the given set of TGDs) by some other atom of body$(q)$ are eliminated. Roughly speaking, the elimination of an atom from the body of a query implies the avoidance of the construction of redundant queries during the rewriting process. Thus, this step greatly reduces the number of BCQs in the perfect rewriting. Before going into the details, let us first introduce some necessary technical notions.

**Definition 3 (Dependency Graph):**

Consider a set $\Sigma$ of TGDs over a schema $\mathcal{R}$. The dependency graph of $\Sigma$ is a labeled directed multigraph $(N, E, \lambda)$, where $N$ is the node set, $E$ is the edge set, and $\lambda$ is a labeling function $E \rightarrow \Sigma$. The node set $N$ is the set of positions of $\mathcal{R}$. If there is a TGD $\sigma \in \Sigma$ such that the same variable appears at position $\pi_b$ in body$(\sigma)$ and at position $\pi_h$ in head$(\sigma)$, then in $E$ there is an edge $e = (\pi_b, \pi_h)$ with $\lambda(e) = \sigma$.

Intuitively speaking, the dependency graph of a set $\Sigma$ of TGDs describes all the possible ways of propagating a term from a position to some other position during the construction of the chase under $\Sigma$. More precisely, the existence of a path $P$ from $\pi_1$ to $\pi_2$ implies that it is possible (but not always) to propagate a term from $\pi_1$ to $\pi_2$. The existence of $P$ guarantees the propagation of a term from $\pi_1$ to $\pi_2$ if, for each pair of consecutive edges $e = (\pi, \pi')$ and $e' = (\pi', \pi'')$ of $P$, where $e$ and $e'$ are labeled by the TGDs $\sigma$ and $\sigma'$, respectively, the atom obtained during the chase by applying $\sigma$ triggers $\sigma'$. To verify whether this holds we need an additional piece of information, the so-called equality type, about the body-atom and the head-atom of each TGD that occurs in $P$.

**Definition 4 (Equality Type):**

Consider an atom $\sigma$ of the form $r(t_1, \ldots, t_n)$, where $n \geq 1$. The equality type of $\sigma$ is the set of equalities

$$\{r[i] = r[j] \mid t_i, t_j \notin \Delta_c \text{ and } t_i = t_j\} \cup \{r[i] = c \mid c \in \Delta_c \text{ and } t_i = c\}.$$

We denote the above set as $eq(\sigma)$.

It is straightforward to see that, given a pair of TGDs $\sigma$ and $\sigma'$, if $eq(\text{body}(\sigma')) \subseteq eq(\text{head}(\sigma))$, then the atom obtained by applying $\sigma$ during the construction of the chase triggers $\sigma'$. Consequently, the existence of a path $P$ (as above) guarantees the propagation of a term from $\pi_1$ to $\pi_2$ if, for each pair of consecutive edges $e$ and $e'$ of $P$ which are labeled by $\sigma$ and
Consider a BCQ \( q \) over a schema \( \mathcal{R} \), and a set \( \Sigma \) of TGDs over \( \mathcal{R} \). Let \( a \) and \( b \) be atoms of \( \text{body}(q) \), where \( \{t_1, \ldots, t_n\} \), for \( n \geq 0 \), is the set of shared variables and constants that occur in \( b \). Also, let \( G_\Sigma \) be the dependency graph of \( \Sigma \). We say that \( a \) covers \( b \) w.r.t. \( \Sigma \), written as \( a \prec_\Sigma b \), if for each \( i \in [n] \): (i) the term \( t_i \) occurs also in \( a \), and (ii) if \( t_i \) occurs in \( a \) and \( b \) at positions \( \Pi_{a,i} \) and \( \Pi_{b,i} \), respectively, then, for each \( \pi \in \Pi_{b,i} \), in \( G_\Sigma \) there exists a path \( \pi_1, \pi_2, \ldots, \pi_k \), where \( k \geq 2 \), such that:

1. \( \pi_1 \in \Pi_{a,i} \) and \( \pi_k = \pi \),
2. for each \( j \in [k - 1] \), there exists \( \sigma_j \) such that \( \lambda(e_{i,j}) = \sigma_j \), where \( e_{i,j} = (\pi_1, \pi_{i,j+1}) \), and \( \text{eq}(\text{body}(\sigma_{j+1})) \subseteq \text{eq}(\text{head}(\pi_j)) \), and
3. \( \text{eq}(\text{body}(\sigma_1)) \subseteq \text{eq}(a) \).

Condition (i) ensures that by removing \( b \) from \( q \) we do not loose any constant, and also all the joins between \( b \) and the other atoms of \( \text{body}(q) \), except \( a \), are preserved. Condition (ii) guarantees that the atom \( b \) is logically implied (w.r.t. \( \Sigma \)) by the atom \( a \), and therefore can be eliminated. The next technical lemma follows.

**Lemma 3:** Let \( q \) be a BCQ over a schema \( \mathcal{R} \), and \( \Sigma \) be a set of linear TGDs over \( \mathcal{R} \). Suppose that \( a \prec_\Sigma b \), where \( a, b \in \text{body}(q) \), and \( q' \) is the BCQ obtained from \( q \) by eliminating the atom \( b \). Then, \( I \models q \) if \( I \models q' \), for each instance \( I \) that satisfies \( \Sigma \).

Given a BCQ \( q \) and a set of TGDs \( \Sigma \), for each atom \( a \in \text{body}(q) \), let \( \text{cover}(q, \Sigma, a) \) be the set of atoms of \( q \) that cover \( a \) w.r.t. \( \Sigma \), i.e., \( \text{cover}(q, \Sigma, a) = \{ b \mid b \in \text{body}(q) \text{ and } b \text{ covers } a \\} \text{ w.r.t. } \Sigma \). When \( q \) and \( \Sigma \) are obvious from the context, we shall denote the above set as \( \text{cover}(a) \). Now, by exploiting the “cover set” of each atom in the body of \( q \), we can compute the set of atoms of \( \text{body}(q) \) that can be eliminated in order to obtain a logically equivalent query (w.r.t. \( \Sigma \)) with fewer number of atoms in its body. This can be done by applying the function \( \text{Eliminate} \), which accepts as input a BCQ \( q \) and a set of TGDs \( \Sigma \). Assuming that \( q \) is of the form \( q() \leftarrow a_1, \ldots, a_n \), \( \text{Eliminate}(q, \Sigma) \) works as follows:

\[
Q := \emptyset \\
\text{for each } i := n \text{ to } 1 \text{ do} \\
\quad \text{if } \text{cover}(a_i) \neq \emptyset, \text{ then } Q := Q \cup \{a_i\} \\
\quad \text{for each } a \in \text{body}(q) \setminus Q \\
\quad \quad \text{cover}(a) := \text{cover}(a) \setminus \{a_i\} \\
\text{return } Q.
\]

We are now ready to describe how query elimination works. During the execution of the rewriting algorithm \( \text{TGD-rewrite} \) (see Algorithm 1), before applying the factorization step we apply the so-called elimination step. In particular, before calling the function \( \text{Factorize}(q, \sigma) \), in order to compute the factorized query \( q' \), we first compute the query \( q^* \) obtained by eliminating from \( \text{body}(q) \) the set of atoms \( \text{Eliminate}(q, \Sigma) \). Then, instead of giving as input to the function \( \text{Factorize} \) the query \( q \), we give the query \( q^* \). An example of query elimination follows.

**Example 6 (Query Elimination):**

Consider the set \( \Sigma \) of TGDs of Example 5, and the BCQ

\[
q() \leftarrow p(A, B), r(A, B, C), s(A, A, D) .
\]

Based on the Definition 5, it is an easy task to verify that \( \text{cover}(a) = \emptyset \), \( \text{cover}(b) = \{a\} \) and \( \text{cover}(c) = \emptyset \). Therefore, the output of the function \( \text{Eliminate}(q, \Sigma) \) is the singleton set \( \{b\} \). Consequently, by applying the elimination step we get the BCQ \( q() \leftarrow p(A, B), s(A, A, D) \).

As already mentioned, the fact that an atom \( a \) covers some atom \( b \) means that \( b \) is logically implied (w.r.t. the given set...
of TGDs) by \(a\). However, as shown by the next example, this fact is not also sufficient for the implication of \(b\) by \(a\).

**Example 7 (Atom Implication):**
Consider the set \(\Sigma\) of TGDs of Example 5, and the BCQ \(q\)

\[
q() \leftarrow r(A, A, c), p(A, A) \quad \frac{1}{2}
\]

where \(c\) is a constant of \(\Delta_c\). Observe that \(a\) does not cover \(b\) since, despite the existence of the paths \(r[1][s][1][p][1]\) and \(r[2][s][3][p][2]\) in the dependency graph of \(\Sigma\), \(eq(body(\sigma_3)) \not\subseteq eq(head(\sigma_2))\). However, \(b\) is logically implied (w.r.t. \(\Sigma\)) by \(a\).

In particular, for every instance \(I\) that satisfies \(\Sigma\), if \(I \models a\), which implies that an atom of the form \(r(V, V, c)\) exists in \(I\), then due to the TGDs \(\sigma_2\) and \(\sigma_3\) there exists also an atom \(p(V, V)\), and thus \(I \models b\). Note that such cases are identified by the C&B algorithm [14]. Nevertheless, as already criticized in Section II, this requires to pay a price in the number of queries in the rewritten query.

It is not difficult to see that the function Eliminate runs in quadratic time in the number of atoms of \(body(q)\) (by considering the given set of TGDs as fixed). In particular, to compute the “cover set” of each body-atom of \(q\) we need to consider all the pairs of atoms of \(body(q)\). Note that the problem whether a certain atom \(a\) covers some other atom \(b\) is feasible in constant time since the given set of TGDs (and thus its dependency graph) is fixed.

The following result implies that the rewriting algorithm TGD-\(\text{rewrite}^*\), obtained from TGD-\(\text{rewrite}\) by applying the additional step of elimination, is still sound and complete.

**Theorem 4:** Let \(\mathcal{R}\) be a relational schema. Consider a BCQ \(q\) over \(\mathcal{R}\), a database \(D\) for \(\mathcal{R}\), and a set \(\Sigma\) of linear TGDs over \(\mathcal{R}\). Then, \(D \models \text{TGD-\(\text{rewrite}^*\)}(\mathcal{R}, \Sigma, q) \iff D \cup \Sigma \models q\).

**Proof (Sketch).** This result follows from the fact that the algorithm TGD-\(\text{rewrite}\) is sound and complete under linear TGDs (see Theorem 1) and Lemma 3.

It is important to clarify that the above result does not hold if we consider arbitrary TGDs (as in Theorem 1). This is because the proof of Lemma 3, which states that atom coverage implies logical implication (w.r.t. the given set of TGDs), is based heavily on the linearity of TGDs. Termination of TGD-\(\text{rewrite}^*\) follows immediately from the fact that TGD-\(\text{rewrite}\) terminates under linear TGDs (see Theorem 2).

**VII. IMPLEMENTATION AND EXPERIMENTAL SETTING**
A prototype implementation of TGD-\(\text{rewrite}^*\) is presented in the Nyaya system\(^7\). Note that this system is based on the IRIS Datalog Engine\(^8\) extended to support the FO-rewritable fragments of the Datalog\(^+\) family. Both IRIS and our extension are implemented in Java.

Since TGD-\(\text{rewrite}^*\) is designed for reasoning over ontologies with large ABoxes, we put ourselves in a similar experimental setting such that of [18] and [19]. Thus, we focus on DL-Lite\(\mathcal{X}\) ontologies with a varying number of axioms. The queries under consideration are based on canonical examples used in the research projects where these ontologies have been developed. STOCKEXCHANGE (S) is an ontology for representing the domain of financial institutions of the European Union. UNIVERSITY (U) is a DL-Lite\(\mathcal{X}\) version of the LUBM Benchmark\(^9\), developed at Lehigh University, and describes the organizational structure of universities. ADOLENA (A) (Abilities and Disabilities OntoLogy for ENhancing Accessibility) is an ontology developed for the South African National Accessibility Portal, and describes abilities, disabilities and devices.

In order to compare different rewriting techniques an important metric is the size of the perfect rewriting, i.e., the number of queries in the rewritten query. We compared our implementation with two other rewriting-based query answering systems for FO-rewritable ontologies: QuOnto\(^10\), based on [4] and developed by the University of Rome La Sapienza, and Requiem\(^11\), based on [18] and developed by the Knowledge Representation group of the University of Oxford. Table II shows a small portion of our experiments\(^12\). By QO and RQ we refer to the QuOnto and Requiem systems, respectively, while NY and NY* refer to Nyaya with factorisation and Nyaya with both factorisation and query elimination, respectively. All the tests have been performed on an Intel Core 2 Duo Processor at 2.50 GHz and 4GB of RAM. The OS is Ubuntu Linux 9.10 carrying a Sun JVM Standard Edition with maximum heap size set at 2GB of RAM.

**TABLE II  EVALUATION OF TGD-\(\text{rewrite}\) WITH QUERY ELIMINATION (TGD-\(\text{rewrite}^*\))**

<table>
<thead>
<tr>
<th>Size</th>
<th>QO</th>
<th>RQ</th>
<th>NY</th>
<th>NY*</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>783</td>
<td>402</td>
<td>402</td>
<td>249</td>
</tr>
<tr>
<td>Q2</td>
<td>1812</td>
<td>103</td>
<td>103</td>
<td>94</td>
</tr>
<tr>
<td>Q3</td>
<td>4763</td>
<td>104</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>Q4</td>
<td>7251</td>
<td>492</td>
<td>492</td>
<td>456</td>
</tr>
<tr>
<td>Q5</td>
<td>66088</td>
<td>624</td>
<td>624</td>
<td>624</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Q2</td>
<td>287</td>
<td>148</td>
<td>148</td>
<td>1</td>
</tr>
<tr>
<td>Q3</td>
<td>1260</td>
<td>224</td>
<td>224</td>
<td>4</td>
</tr>
<tr>
<td>Q4</td>
<td>5364</td>
<td>1628</td>
<td>1628</td>
<td>2</td>
</tr>
<tr>
<td>Q5</td>
<td>9245</td>
<td>2960</td>
<td>2960</td>
<td>2</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q2</td>
<td>204</td>
<td>160</td>
<td>160</td>
<td>1</td>
</tr>
<tr>
<td>Q3</td>
<td>1194</td>
<td>480</td>
<td>480</td>
<td>2</td>
</tr>
<tr>
<td>Q4</td>
<td>1632</td>
<td>960</td>
<td>960</td>
<td>2</td>
</tr>
<tr>
<td>Q5</td>
<td>11487</td>
<td>2880</td>
<td>2880</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^7\)http://mais.dia.uniroma3.it/Nyaya/Home.html
\(^8\)http://www.iris-reasoner.org/
\(^9\)http://swat.cse.lehigh.edu/projects/lubm/
\(^10\)http://www.dis.uniroma1.it/quonto/
\(^11\)http://www.comlab.ox.ac.uk/projects/requiem/home.html
\(^12\)The full comparison is available in the extended version of this paper on the Nyaya’s Web site.
As it can be seen, query elimination provides a substantial advantage in terms of the size of the perfect rewriting for the real-world ontologies A, U and S. In particular, for the queries denoted as Q2 in U and S, our procedure eliminates all the redundant atoms in the input query, and drastically reduces the number of queries in the final rewriting.

**VIII. FUTURE WORK**

We plan to investigate rewriting and optimization techniques for sticky-join sets of TGDs, and alternative forms of rewriting such as positive-existential queries. We also plan to develop improved techniques for rewriting an ontological query into a non-recursive Datalog program, rather than into a union of conjunctive queries (recall the discussion in Section II). While the current approaches yield exponentially large non-recursive Datalog programs, it is possible to rewrite queries and T-Boxes into non-recursive Datalog programs whose size is simultaneously polynomial in the query and the T-Box. This will be dealt in a forthcoming paper.

**Acknowledgments.** G. Gottlob’s work was funded by the European Research Council under the European Community’s Seventh Framework Programme (FP7/2007–2013)/ERC grant no. 246858 – DIADEM. Gottlob gratefully acknowledges a Royal Society Wolfson Research Merit Award. G. Orsi and G. Gottlob also acknowledge the Oxford Martin School - Institute for the Future of Computing. A. Pieris’ work was funded by the EPSRC project “Schema Mappings and Automated Services for Data Integration and Exchange” (EP/E010865/1).

**REFERENCES**


