Enhancing Ontological Query-Rewriting via Parallelization

Discussion Paper

Heba M. Wagih[1], Marco Calautti[2]

[1]Department of Information Engineering and Computer Science, University of Trento, Italy
[2]Department of Computer Science, University of Milan, Italy

Abstract

Ontological databases have become the distinguished knowledge bases for today's systems, which consist of an extensional database and an ontology. Several ontological languages have been proposed to model domain knowledge such as Datalog[∃] which extends Datalog with existential quantification. This addition has enriched the language capabilities in capturing complex domain knowledge, however, it makes reasoning tasks, such as query answering, undecidable. Identifying decidable fragments of Datalog[∃] is an important approach to deal with the above issues, where prominent examples are fragments supporting so-called query rewritability. Several rewriting algorithms have been introduced; however, these algorithms are executed in a sequential fashion. In this paper we discuss a methodology that allows existing rewriting algorithms to exploit multi-core architectures, and show how to employ this methodology to implement a parallel version of an existing query rewriting algorithm.

Keywords

Datalog, Ontological query answering, Query rewriting

1. Introduction

Knowledge representation and reasoning have gained a lot of interest in the last decade. The so-called ontological database, a.k.a. knowledge base, has evolved to be a prominent approach for managing today's system data. A knowledge base merges an extensional database (e.g., a relational database) with an ontology, i.e., a logical theory encoding domain knowledge. The addition of an ontology on top of a standard database then allows to derive additional (called intensional) knowledge that was never explicit in the database alone.

This is a step forward in enhancing reasoning activities. An ontology is an explicit specification of a conceptualization of an area of interest [1].
Several formal languages have been proposed to model ontologies, among which there are the ones based on Description Logics (DLs) [2], which represents the logical formalism underpinning the OWL standard. Another ontology language that can express complex knowledge is Datalog, a rule-based language, that has been introduced to overcome some limitations of existing database query languages, by the introduction of recursion [3].

SEBD 2023: 31st Symposium on Advanced Database System, July 02–05, 2023, Galzignano Terme, Padua, Italy
hebatalla.hammad@unitn.it (H. M. Wagih); marco.calautti@unimi.it (M. Calautti)

Different Datalog extensions have been proposed [4] to express, among others, equality predicates, disjointness constraints, and existential quantification which have an important role in knowledge representation. Datalog[$\exists$] is a prominent example for an extension of standard Datalog via the addition of existential quantification in the rules defining the ontology. Although this addition has enriched the language capabilities in capturing more complex domain knowledge, however, it makes reasoning tasks, such as query answering, undecidable. The undecidability of the query answering problem over knowledge bases using Datalog[$\exists$] ontologies is one of the main challenges that both the database and knowledge representation communities are facing nowadays. Identifying decidable fragments of Datalog[$\exists$] is one of the main ways to deal with the above issues.
Prominent examples are fragments supporting so-called query rewritability, i.e., Datalog[$\exists$] ontologies for which a given query and the ontology can be compiled to a new query, in particular, to a union of conjunctive queries (UCQ), that can be evaluated directly on the extensional database alone. The result of the evaluation will be the same as if the original query is executed against the extensional database and the ontology rules.

Example 1. *Consider the following Datalog[$\exists$] ontology rule σ which asserts that for each project X in some department Y, there exists a supervisor Z who is assigned to the department Y of the project, and consider the (conjunctive) query (CQ) Q which asks for all projects who have a supervisor assigned in the 'db' department:*

$$\sigma : \forall X \forall Y \, \textit{project}(X), \textit{inDept}(X, Y) \rightarrow \exists Z \, \textit{supervisor}(Z, Y, X)$$

$$Q(B) \text{ :- } \exists A \ \textit{supervisor}(A, db, B).$$

The equivalent rewritten query of the above ontology Σ and query Q, is the following UCQ:

$$\begin{aligned} Q_\Sigma \quad &= \{Q(B) \text{ :- } \exists \, A \ \textit{supervisor}(A, db, B), \\ &\quad Q(B) \text{ :- } project(B), inDept(B, db)\}. \end{aligned}$$

The above query states that an answer (tuple) (B) to the query is a project of the database that is either explicitly *assigned a supervisor at the 'db' department,* or *a project at the 'db' department.*

Several rewritable fragments of Datalog[$\exists$], and corresponding rewriting algorithms, have been introduced in recent years, however, to the best of our knowledge, all such algorithms are only meant to be executed in a sequential manner, and hence do not scale well with large ontologies as in the worst case, query rewriting leads to an exponentially sized new UCQ. This drawback has influenced the database research community to develop new algorithms that can enhance the query rewriting process. In [5], the authors proposed a new rewriting algorithm that is based on previous query rewritings. If a given query was previously written and has been extended with new atoms, instead of starting the rewriting from scratch, the authors suggested using any previous rewriting for the original query and extend it with the rewriting of the new atoms. Other remarkable contributions were proposed in [6, 7] .

Despite the work that has been introduced in this area, however, the sequential execution of these algorithms remains the main bottleneck.

Contributions. The goal of this work is to explore the following research goals:

1. Introduce a new methodology that allows to existing rewriting algorithms to perform parallel computations.
2. By means of the above methodology, define a parallel version of one of the main (sequential) query rewriting algorithms known in the literature.

Organization. The paper is organized as follows: Section 2 introduces basic notation and notions. Section 3 discusses the state-of-the-art regarding query rewriting techniques for Datalog[$\exists$]-based ontologies. The proposed methodology for enhancing query answering via parallelization, together with an algorithmic proposal based on this methodology is discussed in Section 4. Conclusions and directions for future work as presented in Section 5.

2. Preliminaries

Relational Schema: A relational schema R consists of a finite set of predicates (relation names), each with a specific arity. We use Δ, N, and V to denote countably infinite sets of *constants*, *nulls* and *variables*. A *term* is a constant, null or variable. An atomic formula (or simply atom) over a schema R is an expression of the form $p(\bar{t})$, where p is a predicate of R with arity n, and $\bar{t} = t_1, \ldots, t_n$ is a tuple of terms. An atom without variables is called a *fact*. An *instance* is a set of facts, while a *database* is an instance without nulls.

(Unions of) Conjunctive Queries (UCQs): A Conjunctive query (CQ) Q is a rule-based query that corresponds to the select-project-join fragment of relational algebra queries and can be represented in either a first-order logic form or rule-like syntax form [3].

In a rule-like syntax form, a CQ is an expression of the form:

$$Q(\bar{x}) : -R_1(\bar{y}_1), \ldots, R_n(\bar{y}_n),$$

where $Q(\bar{x})$ is an atom, called the *head* of the query, and denoted head(Q), with $\bar{x}$ are the *output variables* of Q, while $R_1(\bar{y}_1), \ldots, R_n(\bar{y}_n)$ is a sequence of atoms, called the *body* of the query; body(Q) denotes the *set* of atoms of the body of Q. A *union of Conjunctive queries (UCQ)* is a set of CQs all having the same head predicate.

Answer of a UCQ over a database: Answers of a CQ Q over an instance I are defined via homomorphisms. A *homomorphism* from a set of atoms A to a set of atoms B is a function h mapping terms in A to terms in B, being the identity over the constants. Given an instance I, a conjunctive query Q with output variables $\bar{x}$, and a tuple of constants and nulls $\bar{t}$, we say that $\bar{t}$ is an answer to Q over I if there is a homomorphism h from the body of Q to I such that $h(\bar{x}) = \bar{t}$.

Datalog[$\exists$] Ontologies and Knowledge Bases: A Datalog[$\exists$] rule, a.k.a. *tuple generating dependency (TGD)* is an expression of the form:

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

where each $R_i(\bar{x}_i)$ is an atom over the tuple of variables $\bar{x}_i$, with $\bar{z} \subseteq \bar{x}_0$ In particular, $R_0(\bar{x}_0)$ is the *head*, and $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$ is the *body* of the rule.[1] An *ontology* Σ is a finite set of TGDs.

[1]For simplicity, we often omit the universal quantifier from rules.

Models of a Knowledge Base: A *knowledge base* is a pair (D, Σ) of a database D and an ontology Σ. A model of (D, Σ) is an instance I such that D $\subseteq$ I and I satisfies each TGD of Σ; I satisfies a TGD σ if, whenever there exists a homomorphism h from its body to I, then there is an extension h' $\supseteq$ h of h, such that h' is a homomorphism from the head of σ to I.

Ontological Query Answering: Given a knowledge base (D, Σ) and a UCQ $Q(\bar{x}$, the answers to $Q(\bar{x})$ over (D, Σ), denoted ans(Q, D, Σ) is the set of tuples $\bar{t}$ of constants such that $\bar{t} \in Q(I)$, for every model I of (D, Σ) [8].

It is well-known that *ontological query answering*, i.e., the problem of checking whether a given tuple $\bar{t}$ is an answer to a given UCQ over a given knowledge base, is undecidable, unless restrictions on the ontology language are enforced.

3. State of the Art

Datalog[$\exists$] is an extension of classical Datalog that enhances its expressivity by introducing existential quantifiers to its rules. Although this version has enhanced the knowledge representation capabilities of the language, this increase in expressive power makes ontological query answering undecidable. Current approaches to tackle this issue focused on identifying fragments of the Datalog[$\exists$] language, that guarantee that ontological query answering becomes decidable. The most prominent approach is the one of rewritability. An ontology Σ is *rewritable* if for any UCQ Q, it is possible to rewrite Σ and Q to a new UCQ Q' such that, for *every* database D, the answers of Q over the knowledge base (D, Σ) coincide with the answers of Q' over D alone, i.e., ans$(Q, D, \Sigma) = Q'(D)$, for every database D. Such a query Q' is called a *perfect rewriting* of Σ and Q. Different fragments of Datalog[$\exists$] have been introduced in the literature that guarantee ontologies written in such fragments are rewritable, such as *linear* ontologies, *multi-linear* ontologies and *sticky* ontologies. All such fragments are powerful enough to express ontologies written in other important formalisms such as Description Logics [8].

Several efforts in implementing rewriting algorithms for the these fragments have been introduced. One prominent example is the so called XRewrite algorithm [8]. The XRewrite algorithm is a generic rewriting algorithm that can be applied to any rewritable ontology language, although the authors experimentally evaluated the algorithm only for linear and sticky TGD classes. The main idea of the proposed algorithm is to generate a union of conjunctive queries given a conjunctive query CQ as input and a set of TGDs. The algorithm is a backward chaining rewriting algorithm that employes a so-called rewriting step which starts from the body of a given query and exhaustively compares it with each rule head in the set of TGDs to give rise to a new CQ that will become a part of the final rewritten UCQ.

Example 2. *Consider the following TGD σ:*

$$\textit{project}(X), \textit{inDept}(X, Y) \rightarrow \exists Z\, \textit{supervisor}(Z, Y, X)$$

and the query

$$Q(B) \text{ :- } \textit{supervisor}(A, db, B)$$

taken from Example 1.

The rewriting step starts with finding a match for the head of σ, i.e., head(σ), and (a subset of) the body of Q, body(Q). The match in this case is for supervisor(A, db, B) and supervisor(Z,Y,X), which is {$X \rightarrow$ B,$Y \rightarrow$ db, $Z \rightarrow$ A}. Thus, the atom supervisor(A, db, B) in in the body of Q can be replaced with the body of σ.[2] The result of the rewriting step is a new CQ Q.

$$Q(B) \text{ :- } project(B), inDept(B, db).$$

Applying the rewriting step without considering some important constraints could lead to unsound or incomplete rewritings. To overcome such problems, two conditions are required before a rewriting step can be performed, namely the applicability condition, which, roughly, guarantees that the rewriting step does not produce a new query obtained by means of joining values coming from existential variables with constants, and factorization, which is an intermediate operation that simplifies a given query, so that the applicability condition can be applied.

The XRewrite algorithm is thus an exhaustive application of rewriting step and factorization, until no new queries can be produced, and the result of the algorithm is the union of all queries produced. The key property of the algorithm is that it always terminates, whenever the input ontology is rewritable. The authors introduced some preliminary work on making this algorithm run in parallel, but this has been achieved only for Linear TGDs. The authors achieved the parallelization by decomposing the input query into smaller chunks and rewrite each of these chunks independently, using the XRewrite algorithm and finally merge the rewriting of each chunk into the final rewriting.

Another algorithm that exploits similar ideas to the XRewrite is the PURE rewriting algorithm [9]. The PURE rewriting algorithm merges the rewriting step and factorization in a single operator, called *piece-wise unifier* and apply a pruning strategy that is able to remove queries obtained in the rewriting process, that are subsumed by more general queries.

4. Ontology Reasoning via Parallelization

To the best of our knowledge, all existing rewriting algorithms are essentially computing sequentially, and thus do not scale well with large ontologies. The research problem to be addressed is thus to identify methodologies for parallelization and then, extend the preceding algorithms with these approaches. To apply parallelization, we can exploit an idea that was initially introduced in [8] which is the decomposition of the query. In [8], the authors proposed the decomposition of the input query into smaller parts (subqueries) that can then be rewritten independently. We present query decomposition with an example.

Example 3. *Consider the following TGD σ and CQ Q.*

$$person(X) \rightarrow \exists Z\ hasFather(X, Z)$$

[2]Variables in the body of σ that do not appear in its head are assumed to be replaced with fresh variable names, to avoid clashes with existing variables in the query.

$$Q(X) \text{ :- } \mathit{hasFather}(X, Y), \mathit{employee}(Y)$$

.

Note that due to the applicability condition, the head of σ cannot be unified with the atom $\mathit{hasFather}(X, Y)$ of the query, since Y is a join variable in the query, and this variable Y is going to be mapped to the existential variable of σ.

With the above in mind, it is clear that to safely decompose the query, we first need to understand which variables in the query have a chance to be unified with an existential variable, at any step of the rewriting algorithm*; atoms of the query mentioning such variables should never be separated, when decomposing the query.*

Example 5. *Consider the following example.*

σ_1*: person*$(X) \rightarrow \exists Z$ *hasFather*(X, Z)
σ_2 *: son*$(X) \rightarrow$ *male*(X)
σ_3 *: father*$(X) \rightarrow$ *male*(X)
σ_4*: son*$(X) \rightarrow \exists Z$ *hasFather*(X, Z)
σ_5 *: daughter*$(X) \rightarrow \exists Z$ *hasFather*(X, Z)
σ_2 *: daughter*$(X) \rightarrow$ *female*(X)

Consider also the following conjunctive query Q asking for all (a, b), where a is a male and has father b.

$$Q(A, B) \quad \text{:-} \quad \mathit{male}(A), \mathit{hasFather}(A, B).$$

After inspecting the rules in Σ, it is easy to verify that there is no join variable occurring in Q that has a chance to be unified with an existential variable during the rewriting process, therefore, we can safely decompose Q. For example, as the following 2 subqueries[3]:

$$\begin{array}{lll} Q_1(A) & \text{:-} & \mathit{male}(A) \\ Q_2(A, B) & \text{:-} & \mathit{hasFather}(A, B) \end{array}$$

We then use a so-called reconciliation rule *to encode the original query as the conjunction of the above to sub-queries:*

$$\rho : Q(A, B) \text{ :- } Q_1(A), Q_2(A, B),$$

that intuitively says that Q is the query obtained by computing the join of the queries $Q_{1,\Sigma} and Q_{2,\Sigma}$.

The simplest form of parallelization one can employ is thus the one where given a UCQ Q and an ontology Σ, we first decompose Q into subqueries Q_1,…,Q_m, and then employ an off-the-shelf sequential rewriting algorithm over each Q_i w.r.t. Σ. Then, by using the reconciliation rule ρ, and the rewritings of each Q_i, one can perform the "unfolding" of ρ, obtainig the final rewritten UCQ. This is the approach employed in [8], which allowed to improve performance when applied to linear TGDs, however, for other types of TGDs as multi-linear, or sticky, it again does not scale well, since in each step of the rewriting, when employing TGDs with multiple atoms in their body, such as multi-linear, a new query with more and more atoms in its

[3]Each subquery Q_i will have as output variables, the subset of the output variables of Q which Q_i mentions together with the variables that also other subqueries mention.

body may be generated. To achieve better results with different types of TGDs, we propose to perform query decompositions not only for the input query, but *internally to the algorithm.* That is, we propose applying the decomposition algorithm in each execution cycle before applying the rewriting. This allows to keep the queries being worked on by the algorithm small, at each iteration cycle, and at the same time allows to employ multiple threads during the *whole* execution of the algorithm, rather than just at the start.

We present a preliminary modification of the XRewrite algorithm using the above approach in Algorithm 1.

Algorithm 1 The modified XRewrite algorithm

Input: a CQ Q over a schema R and a set Σ of TGDs over R
Output: the perfect rewriting of Q w.r.t. Σ
Q_{REW} := {(Q, **r**, **u**)};
repeat
  Q_{TEMP} := Q_{REW};
  foreach (Q, x, **u**) $\in Q_{TEMP}$, where x $\in$ {**r**, **f**} **do**
    ($Q^1, \ldots, Q^m, \rho$) := decompose(Q, Σ);
    for $i = 1$ **to** m **do in parallel**
      Q^i_{REW} := $\emptyset$;
      foreach $\sigma \in \Sigma$ **do**
        /* rewriting step */
        foreach S $\subseteq$ body(Q^i) such that σ is applicable to S **do**
          Let γ be the MGU for S,Q^i;
          Construct Q' by replacing S with body(σ) in Q^i, and
            apply γ to all variables of Q';
          Q^i_{REW} := $Q^i_{REW} \cup$ (Q', **r**, **u**);
        end
        /* factorization step */
        foreach S $\subseteq$ body(Q^i) which is factorizable w.r.t. σ **do**
          Let γ be the MGU for the atoms in S;
          Construct Q' by applying γ to the variables of body(Q);
          Q^i_{REW} := $Q^i_{REW} \cup$ (Q', **f**, **u**);
        end
      end
    end
    /* query Q is now explored */
    Q_{REW} := ($Q_{REW} \setminus$ {(Q, x, **u**)}) $\cup$ {(Q, x, **e**)};

    /* unfolding of the rewritings */
    Q_Σ := unfold($Q^1_{REW}, \ldots, Q^m_{REW}, \rho$);

    foreach Q in Q_Σ obtained from queries all marked with **r** **do**
      if there is no (Q", **r**, *) $\in Q_{REW}$ which is equal to Q,
        up to variable renaming **then**
        Q_{REW} := $Q_{REW} \cup$ {(Q, **r**, **u**)};
      end
    end
    foreach Q in Q_Σ obtained by using a query marked with **f** **do**
      if there is no (Q", *, *) $\in Q_{REW}$ which is equal to Q,
        up to variable renaming **then**
        Q_{REW} := $Q_{REW} \cup$ {(Q, **f**, **u**)};
      end
    end
  end
until Q_{TEMP} = Q_{REW};
Q_{FIN} := {Q | (Q, **r**, **e**) $\in Q_{REW}$};
return Q_{FIN}

The algorithm keeps track of all intermediate rewritten queries in a set Q_{REW}, where each

query is marked with two flags: the first is either **r** or **f**, meaning that the query is going to be part of the final rewriting, or it is the result of an intermediate factorization, while the second is either **u** or **e** meaning the query is unexplored, and thus needs to be processed, or it has been already processed (explored). The main idea is to initialize Q_{REW} with the input query, and then, iteratively do the following: decompose the currently worked-on query extracted from Q_{REW}, via the decomposition approach explained earlier. Then, for each subquery Q^i, an independent thread is used to apply the rewriting and the factorization steps to Q^i exhaustively, as in the original XRewrite algorithm, and the resulting UCQ obtained from each subquery Q^i is stored in a dedicated set Q^i_{REW}. The algorithm then marks the worked-on query as explored, and the results are then joined together with the unfolding operation, obtaining a UCQ Q_Σ whose elements (CQs) need to be added to the global set Q_{REW}. In the last part of the algorithm, if a CQ in Q_Σ has been obtained by combining only queries marked with **r**, i.e., as "to be rewritten", then this CQ will be part of the final rewriting, and thus it is marked with **r** in the set Q_{REW}. If instead one of the queries used to construct the CQ was marked with **f**, i.e., "as being just an intermediate query obtained by factorization", then we mark the CQ with **f** in the set Q_{REW} as it means it needs further processing.

Correctness and Termination. To show that the algorithm is correct, and terminates for every ontology Σ that is rewritable, we extend the proofs of correctness and of termination employed in [8]. In particular, the correctness follows from the correctness of the original XRewrite algorithm and the fact that the unfolding of the reconciliation rule ρ of a CQ Q w.r.t. the UCQs that rewrite each component of Q coincides with the perfect rewriting of Q.

Regarding the termination, this follows from the termination of the original XRewrite algorithm, and the fact that the unfolding of the reconciliation rule ρ of a CQ Q w.r.t. the UCQs that rewrite each component of Q coincides with the perfect rewriting of Q, and thus does not introduce more UCQs w.r.t. the ones introduced by sequential version of XRewrite.

5. Next Steps

In this paper, we considered the problem of Ontological Query Answering via query rewriting, for Datalog[$\exists$] ontologies. That is, given a Datalog[$\exists$] ontology Σ and a query Q, compute a UCQ Q_Σ such that the answers to Q over the knowledge base (D, Σ) coincide with the answers of Q_Σ over D, for every database D. We studied the problem from a theoretical perspective, and identified state of the art algorithms that would greatly benefit from parallel implementations. We highlighted a preliminary proposal in this direction by adapting the (sequential) XRewrite algorithm from [8] by embedding parallel computation in the rewriting process, by means of query decomposition techniques. For future research we plan to implement the presented algorithm, and develop a benchmark on real-world ontologies, in a similar spirit to other efforts such as [10]. We would use this benchmark to understand and verify the impact of the new algorithms in different scenarios, and compare them with existing approaches, including ones based on different techniques, such as via the terminating chase (e.g., see [11, 12]). Further direction for future work would be to optimize the query rewriting by exploiting the presence in the database schema of integrity constraints such as standard database dependencies [13] as well as more advanced constraints, such as Active Integrity Constraints [14] and Preferences [15].

References

[1] T. R. Gruber, A translation approach to portable ontology specifications, Knowledge Acquisition 5 (1993) 199–220. doi:https://doi.org/10.1006/knac.1993.1008.

[2] F. Baader, D. Calvanese, D. Mcguinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook: Theory, Implementation, and Applications, 2007.

[3] M. Arenas, P. Barcelo, L. Libkin, W. Martens, A. Pieris, Principles of databases. (2021). doi:https://github.com/pdm-book/community.

[4] G. Gottlob, G. Orsi, A. Pieris, M. Simkus, Datalog and its extensions for semantic web databases (2012). doi:7487.10.1007/978-3-642-33158-9_2.

[5] T. Venetis, G. Stoilos, G. Stamou, Query extensions and incremental query rewriting for owl 2 ql ontologies, Journal on Data Semantics 3 (1) (2013) 1–23.

[6] A. Chortaras, D. Trivela, G. B. Stamou, Optimized query rewriting for owl 2 ql, In Proceedings of the 23rd International Conference on Automated Deduction (2011) 192–206.

[7] S. Kikot, R. Kontchakov, M. Zakharyaschev, Conjunctive query answering with owl 2 ql, In Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (2012 a).

[8] G. Gottlob, G. Orsi, A. Pieris, Query rewriting and optimization for ontological databases, ACM Transactions on Database Systems (2014). doi:39.10.1145/2638546.

[9] M. König, M. Leclere, M.-L. Mugnier, M. Thomazo, Sound, complete and minimal ucq-rewriting for existential rules, Semantic Web (2013). doi:6.10.3233/SW-140153.

[10] M. Calautti, M. Console, A. Pieris, Benchmarking approximate consistent query answering, 2021.

[11] M. Calautti, A. Pieris, Semi-oblivious chase termination: The sticky case, Theory Comput. Syst. 65 (2021).

[12] M. Calautti, G. Gottlob, A. Pieris, Non-uniformly terminating chase: Size and complexity, in: PODS, 2022.

[13] S. Abiteboul, R. Hull, V. Vianu, Foundations of databases, Addison-Wesley (1995).

[14] M. Calautti, L. Caroprese, S. Greco, C. Molinaro, I. Trubitsyna, E. Zumpano, Existential active integrity constraints, Expert Systems with Applications 168 (2021).

[15] M. Calautti, S. Greco, C. Molinaro, I. Trubitsyna, Preference-based inconsistency-tolerant query answering under existential rules, Artificial Intelligence 312 (2022).