QueryPIE: Hybrid Reasoning With The OWL RL Rules

Jacopo Urbani a,*, Robert Piro b Frank van Harmelen a Henri Bal a

a Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
Email: {jacopo,frankh,bal}@cs.vu.nl
b Department of Computer Science, University of Oxford, United Kingdom
Email: robert.piro@cs.ox.ac.uk

Abstract. Both materialization and backward-chaining as different modes of performing inference have complementary advantages and disadvantages. Materialization enables very efficient responses at query time, but at the cost of an expensive up front closure computation, which needs to be redone every time the knowledge base changes. Backward-chaining does not need such an expensive and change-sensitive precomputation, and is therefore suitable for more frequently changing knowledge bases, but has to perform more computation at query time.

Materialization has been studied extensively in the recent semantic web literature, and is now available in industrial-strength systems. In this work, we focus instead on backward-chaining, and we present a general hybrid algorithm to perform efficient backward-chaining reasoning on very large RDF datasets.

To this end, we analyze the correctness of our algorithm by proving its completeness using the theory developed in deductive databases and we introduce a number of techniques that exploit the characteristics of our method to execute efficiently the OWL RL rules. These techniques reduce the computation and hence improve the response time by reducing the size of the generated proof tree and the number of duplicates produced in the derivation.

We have implemented these techniques in a experimental prototype called QueryPIE and present an evaluation on both realistic and artificial datasets of a size that is between five and ten billion of triples. The evaluation was performed using one machine with commodity hardware and it shows that (i) with our approach the initial precomputation takes few minutes against the hours (or even days) necessary for a full materialization and that (ii) the remaining overhead introduced by reasoning still allows single pattern queries to be processed with an interactive response time. To the best of our knowledge our method is the first that demonstrates complex rule-based reasoning at query time over an input of several billion triples and it takes a step forward towards truly large-scale reasoning by showing that complex and large-scale OWL inference can be performed without an expensive distributed hardware architecture.

1. Introduction

The amount of RDF data available on the Web calls for RDF applications that can process this data in an efficient and scalable way.

One of the advantages of publishing RDF data is that applications are able to infer implicit information by applying a reasoning algorithm on the input data. To this end, a predefined set of inference rules which is complete w.r.t. some underpinning logic, can be applied in order to derive additional data.

Several approaches that perform rule-based inference were presented in literature. State of the art methods apply rules in a forward-chaining fashion, so that all the possible derivations are produced and stored with the original input. These methods exhibit good scalability because they can efficiently exploit computational parallelism. However, they have several disadvantages that compromise their use in real-world scenarios. First, they cannot efficiently deal with small incremental updates since they have to compute the com-
plete materialization anew. Second, they become inefficient if the user is interested in only a small portion of the entire input because forward-chaining needs to calculate the entire derivation.

Unlike forward-chaining, backward-chaining applies only inference rules depending on a given query. In this case, the computations required to determine the rules that need to be executed often become too expensive for interactive applications. Thus, it has until now been limited to either small datasets (usually in the context of expressive DL reasoners) or weak logics (RDFS inference).

In this paper we propose a method which materializes a fixed set of selected queries, before query time, whilst applying backward chaining during query time. This hybrid approach is a trade off between a reduction in rule applications at query time and a small, query independent computation of data ante query time. Our method relies on a backward-chaining algorithm to calculate the inference which exploits such partial materialization and the parallel computing power of modern architectures.

We thus have to tackle several problems. The first is to show that our reasoning algorithm is correct, i.e. it terminates, is sound and complete. We shall argue that the correctness holds for any positive Datalog program and is not dependent on a particular ruleset.

For the evaluation, however, we will apply these techniques to the OWL RL ruleset in order to exploit the advantages of the latest standard OWL language on a large scale. We address some crucial challenges that arise with OWL RL and propose a set of novel optimizations that substantially improve the computation and hence the execution time.

We have implemented these techniques in an experimental prototype called QueryPIE, and we have tested the performance using artificial and realistic datasets of a size between five and ten billion triples. The evaluation shows that we are able to perform OWL reasoning using one machine equipped with commodity hardware which keeps the response time often under the second.

This paper is a revised and improved version of our initial work that was presented in [18]. More specifically, it extends the initial version that targets the $pD^*$ fragment to support the OWL RL rules, which are officially standardized by the community. Also, it provides a theoretical analysis of the approach proving its correctness w.r.t. the considered ruleset and presents an improved explanation and evaluation over larger datasets.

The remainder of this paper is organized as follows: in Section 2 we introduce the reader to the problem of large scale reasoning and present at high level the main idea behind hybrid reasoning. The purpose of this section is to familiarize the reader with frequently used concepts in this paper.

In Section 3 we will describe the backward-chaining algorithm that is used within our method to calculate the inference. Section 4 formalizes the precomputation algorithm of hybrid reasoning and proves its correctness. Next, in Section 5, we focus on the execution of the OWL RL ruleset presenting a series of optimizations to improve the performance on a large input. In Section 6 we present an evaluation of our approach using single pattern queries on both realistic and artificial data. In Section 7 we report on related work. Finally, Section 8 concludes and gives some directions for future work.

2. Hybrid reasoning: Overview

In principle there are two different approaches to infer answers in a database with a given ruleset: One is to compute the complete extension of a database under some given ruleset (Datalog program) before query time and the other is to infer only the necessary entries needed to yield a complete answer from the ruleset on-demand, i.e. at query time.

The former’s advantage is that querying reduces, after the full materialisation, to a mere lookup in the database and is therefore very fast compared to the latter approach, where for each answer a proof-tree has to be built.

On the other hand, if the underlying database changes frequently, ex-ante materialisation has a severe disadvantage as the whole extension must be recomputed with each update. In this case, an on-demand approach has a clear advantage.

The approach presented in this paper positions itself inbetween: the answers for a carefully chosen set of queries are materialized before query time and added to the database. Answers to queries later posed by the user are inferred at query time.

Traditionally, each approach has been associated with an algorithmic method to retrieve the results: Backward chaining was specifically aimed at on-demand retrieval of answers, only materialising as little information as necessary to yield a complete set of answers, whilst forward chaining applies the rules of the given ruleset until the closure is reached.
Since we want to avoid complete materialization of the database, and therefore are only interested in specific answers, we use backward chaining in both instances: we use backward chaining to materialize only the necessary information for the carefully chosen queries which we then add to the database, and we use backward chaining to answer the user queries.

To this end, we introduce a backward-chaining algorithm which exploits parallel computing power and the fact that some triple patterns are pre-materialized to improve the performance. For example, if one of these pre-materialized queries is requested at query-time, the backward-chaining algorithm does not need to build the proof-tree, but a lookup suffices. In case the pre-materialized patterns frequently appear at user query-time, such optimization is particularly effective.

To give an idea, how this works, consider the following

**Example 1** Consider the two following rules that are Datalog translations of rules from the OWL RL ruleset:

\[
T(a, p1, b) \leftarrow T(p, SPO, p1) \land T(a, p, b)
\]

\[
T(x, SPO, y) \leftarrow T(x, SPO, w) \land T(w, SPO, y)
\]

where we use from now on the abbreviations\(^1\) EQC and SCO for reasons of space and \(a, b, p, p1, x, y, w\) are variables.

Assume we want to suppress the unfolding of all atoms of the form \(T(x, SPO, y)\), modulo variable renaming. We can replace each atom by some new atom, using an extensional database predicate (edb), i.e. a predicate which does not appear in the head of any rule, say \(S\), rendering it into just a lookup in the database. After the substitution, Example 1 would become:

\[
T(a, p1, b) \leftarrow S(p, SPO, p1) \land T(a, p, b)
\]

\[
T(x, SPO, y) \leftarrow S(x, SPO, w) \land S(w, SPO, y)
\]

Clearly, the two programs in Example 1 do not yield the same answers for \(T\) anymore. To restore this equality we need to calculate all \(T(x, SPO, y)\)-triples and add them to the auxiliary relation named \(S\) in the database. In our example this would mean that \(S\) contains the transitive closure of all \(T(x, SPO, y)\)-triples which are inferrable under the ruleset in the database.

Notice, that \(T(x, SPO, y)\)-triples can also be derived with the first rule if \(p1 = SPO\). Furthermore, if \(S\) indeed contains the transitive closure of all \(T(x, SPO, y)\)-triples the second rule can be rewritten as \(T(x, SPO, y) \leftarrow S(x, SPO, y)\).

Before we formalize this method and show that it is indeed harmless in the sense that everything which could be inferred under the original program can be inferred under the altered program and vice versa, we shall discuss the backward-chaining algorithm we use and how it exploits the pre-materialization.

### 3. Hybrid Reasoning: Backward-chaining

The purpose of the backward-chaining algorithm is to derive all possible triples that are part of a given input query \(Q\), given a database \(D\) and a ruleset \(R\).

Traditionally, users interact with RDF datasets using the SPARQL language [13] where all the triple patterns that constitute the body of the query are joined together according to some specific criteria. In this paper, we do not consider the problem of efficiently joining the RDF data and focus instead on the process of retrieving the triples that are needed for the query. Therefore, we target our reasoning procedure at atomic queries, e.g.,

\[
(?c_1 \text{rdfs:subclassOf} ?c_1)
\]

where question marks indicate variables (cf. Appendix A). Since RDF contains only one generic predicate symbol \(T\), it is not mentioned in the query.

Rules formulated in the SPARQL language like those in the OWL RL ruleset can trivially be rendered into a positive Datalog program as already witnessed in Example 1.

The algorithm that we present is inspired by the well-known algorithm QSQ (Query-subquery) that was first introduced in 1986 which generalizes the SLD-resolution technique [19] by applying it to set of tuples instead of single ones. The variations that we introduce are meant to exploit the computational parallelism that is possible to obtain by using the modern architectures.

The QSQ algorithm recursively rewrites the initial query in many subqueries until no more rewritings can be performed and the subqueries can only be evaluated against the knowledge base.

---

\(^1\)Table 7 in Appendix A contains a list of all the abbreviations that we used in this paper.
Example 2

For example, suppose that our initial query is

\[ T(x, \text{rdf: type, Person}) \]

and that we have a generic input \( D \) and the OWL RL ruleset as \( R \). Initially, the algorithm will determine which rules can produce a derivation that is part of the input query. For example, it could apply the subclass and subproperties inheritance rules (\( \text{cax-sco} \) and \( \text{prp-spo1} \) in the OWL RL ruleset). After it has determined them, it will move to the body of the rules and proceed evaluating them. In case these subqueries will produce some results, the algorithm will execute the rules and return the answers to the upper level. This process creates a tree which is called proof tree. This tree represents all the derivations steps that can be taken to derive the answers that are part of the original query. In Figure 1 we report an example of such tree for our example query.

An important problem of backward-chaining algorithms concerns the execution of recursive rules. Recursive rules and more in general cycles in the proof tree are an important threat since they could create loops in the computation that the algorithm must handle.

The QSQ algorithm guarantees the termination even in presence of recursive rules by memorizing in a global data structure all the subqueries already evaluated and avoiding to make a recursive call with a query if this was already previously done. This means that eventual derivations that require the evaluation of the same query more times cannot be inferred because the algorithm will stop the recursion after one application.

To solve this issue, the algorithm repeats the execution of the query until fix closure. This operation is performed at every recursive call, to ensure that all the bindings for each subquery in the proof tree is correctly retrieved.

It has been proved that the QSQ algorithm is sound and complete [20]. Because of this, we are ensured that with this methodology no derivation will be missed.

The original version of this algorithm is hard to parallelize because it requires a sequential execution to build the proof tree with a depth-first strategy and it exploits the access to a global data structure to remember all the previously derived queries. Therefore, its execution is unable to take advantage of modern multi-core architecture or clusters of several independent nodes.

Because of this, in the next section we will present an adaptation of this algorithm to our specific use case so that it can be more easily parallelized and show that the fundamental properties of termination and completeness are still valid.

3.1. Our approach

We introduced two key differences to improve the parallelization of the computation:

- Instead of constructing the proof tree sequentially using a depth-first strategy, we do it in parallel by applying the rules on separate threads and in an asynchronous manner. But this makes the implementation and the maintenance of the global data structure, used for the caching of previous queries, difficult and inefficient. We hence choose to replace this mechanism with a local data structure to only remember which queries where already executed. While such a choice might lead to some duplicate answers because the same queries can be repeated more times, it allows the computation to be performed in parallel limiting the usage of expensive synchronization mechanisms;

- Because the proof tree is built in parallel, ensuring completeness by having a loop at every recursive call is inefficient since that the same query can appear multiple times on different parts of the tree. Therefore, we replace it with a global loop that is performed only at the root level of the tree and storing at every iteration all the intermediate derivations. In the worst case such choice is not efficient because at every iteration the entire proof tree must be performed another time. However, in the best case this choice reduces the number of rules applications.
Before we present the algorithm, we define the following: Let \( \vec{t} := (t_1, \ldots, t_n) \) and \( \vec{t'} := (t'_1, \ldots, t'_n) \) be tuples with \( t_i, t'_i \in \text{Tuples} \), i.e. each component is either a variable or a constant. Then \( \vec{t} \) is an instance of \( \vec{t'} \), if there is a substitution \( \sigma : \text{Tuples} \rightarrow \text{Tuples} \) such that \( \sigma(c) = c \) for each constant \( c \) and \( \sigma(t_1), \ldots, \sigma(t'_n) = (t_1, \ldots, t_n) \).

Additionally, if \( R(\vec{t}) \) and \( R'(\vec{t'}) \) are atoms we define \( R(\vec{t}) \subseteq R'(\vec{t'}) \) iff \( \vec{t} \subseteq \vec{t'} \) and \( R = R' \).

**Example 3** \((x, \text{TYPE}, \text{SYM}) \subseteq (x, \text{TYPE}, y)\) where \( x, y \) are variables and \text{TYPE} and \text{SYM} are the abbreviation of Table 7 in Appendix A. Also \((x, \text{TYPE}, y) \subseteq (y, \text{TYPE}, x)\).

If \( R(\vec{t}) \subseteq R'(\vec{t'}) \) and \( R'(\vec{t'}) \subseteq R(\vec{t}) \) then \( R(\vec{t}) \) equals \( R'(\vec{t'}) \) up to variable renaming.

We report the algorithm using pseudocode in Algorithm 1. The procedure \text{main} is the main function used to invoke the backward-chaining procedure for a given atomic query \( Q \). It returns the derived answers for \( Q \). The function \text{infer} is invoked with the input query. This function returns all the derived answers for \( Q \) that were calculated by applying the rules using backward-chaining (line 5) and all the intermediate answers that were inferred in the process, saved in the global variable \( \text{Tuples} \). In each loop-pass the latest results in \( \text{Tuples} \) and \( \text{New} \) are checked against the accumulated answers of the previous runs in \( \text{Mat} \) and \( \text{Database} \). If nothing new could be derived the loop terminates.

After this loop has terminated, the algorithm returns \( \text{New} \) (line 7) which contains after the last loop-pass all answers to the input query (cf. line 13) and returns the results.

The function \text{infer} is the core of the backward-chaining algorithm. Using the function \text{lookup}, it first retrieves for the formal parameter \( Q \) all answers which are facts in the database or were previously derived (line 13). After this, it determines the rules that can be applied to derive new answers for \( Q \) (lines 14–15) and calculates the substitution \( \theta \) to unify the head of the applicable rule with the query \( Q \) (line 16). It proceeds with evaluating the body of the rule (lines 16–24) storing in \text{Tuples} and \text{Tmp} the retrieved answers (lines 20–21), and performing the joins necessary according to the rule body (line 22).

The algorithm copies all the substitutions that were derived by applying the rules into the variable \( \text{all_subst} \) and constructs a set of answers using these substitutions \( \theta \) (line 27). This set is then returned to the function caller. After the whole recursion tree has been explored exhaustively, \text{infer} returns control to the function \text{main}, where the derived answers are copied into the variable \( \text{New} \). The process is repeated until the closure is reached.

We will now discuss the correctness of our modified algorithm, which are termination, soundness and completeness.

3.1.1. Termination.

It is easy to verify that the backward-chaining algorithm in Algorithm 1 always terminates. The only two sources for not-termination are (i) the loop in lines 3–7 and (ii) the recursive call in line 20. The first loop will continue until neither \( \text{New} \) nor \( \text{Tmp} \) will contain new answers. This happens latest when every relation
is equal to the cartesian product of its arity over the domain of the database, hence within finitely many steps.

The recursive call in line 23 will be fired only if there is no \( Q’ \) in the set PrevQueries (see condition in line 15) such that \( Q \sqsubseteq Q’ \) and \( Q’ \not\sqsubseteq Q \) (cf. page 5), meaning that \( Q \) equals \( Q’ \) up to variable renaming. However, for similar reasons as before there are, up to variable renaming, only finitely many different atomic queries over the domain of the database and so the recursion will terminate.

3.1.2. Soundness.

The soundness is immediate, as new facts can only be derived through rule-application in the for-loop beginning in line 14. Hence, if \( R(a_1, \ldots, a_n) \) is a fact derived by the function infer, \( R(a_1, \ldots, a_n) \) is a fact in \( P(3) \), the least fix-point model for the Datalog program \( P \) and the database \( \mathcal{D} \). Hence Algorithm 1 is sound.

3.1.3. Completeness.

In order to show the completeness of Algorithm 1, we prove Proposition 2, which holds in particular true for all answers to \( Q \) derived under a given ruleset RuleSet and a database Database. We first show

**Proposition 1** Let \( Q \) be a query for function main and \( R(a_1, \ldots, a_n) \) a fact, which appears in the proof-tree of some fact derived from \( Q \) under RuleSet in Database. Then there is some subsequent non-blocked query \( Q_n \) appearing in the computation of infer\((Q, RuleSet, \emptyset)\), such that \( R(a_1, \ldots, a_n) \) is an answer to \( Q_n \) derived under RuleSet in Database.

**Proof 1** We have to show, that there is a sequence of queries \( Q_0, \ldots, Q_n \) such that

1. \( Q = Q_0 \) and \( R(a_1, \ldots, a_n) \) unifies with \( Q_n \),
2. for each \( i \in \{0, \ldots, n\} \) there is a rule such that \( Q_i \) unifies with the head of some rule \( r \in RuleSet \) and \( Q_{i+1} \) unifies with some body-atom of \( r \),
3. no query is blocked, i.e. there is no subsequence \( Q_1, \ldots, Q_k \) with \( 0 \leq i < k \leq n \) such that \( Q_i \) is up to variable renaming equal to \( Q_k \) (\( Q_i \sqsubseteq Q_k \) and \( Q_k \not\sqsubseteq Q_i \)).

In this case \( \text{infer}(Q, RuleSet, \emptyset) \) will eventually produce the query \( Q_n \) (cf. lines 14-20, of Algorithm 1).

Let \( Q(b_1, \ldots, b_m) \) be the fact which unifies with the input query \( Q \) in which proof-tree \( R(a_1, \ldots, a_n) \) appears. Then there is a sequence of rule applications \( r_0, \ldots, r_n \) such that \( Q(b_1, \ldots, b_m) \) unifies with the head of \( r_0 \), for all \( i \in \{1, \ldots, n\} \) some body-atom \( b_{i,k} \) of \( r_i \) unifies with the head of \( r_{i+1} \) and \( R(a_1, \ldots, a_n) \) unifies with some body-atom \( b_{n,t} \) of \( r_n \).

Since \( Q(b_1, \ldots, b_m) \) was an answer to \( Q \), they unify and so \( Q \) unifies with the head of \( r_0 \) yielding \( \theta_0 := \text{MGU}(Q, r_0) \). For all \( i \in \{1, \ldots, n\} \) the body-atom \( b_{i,k_i} \) of \( r_i \) unifies with the head of \( r_{i+1} \) yielding \( \theta_{i+1} := \text{MGU}(Q_i, r_{i+1}) \) where \( Q_i := \theta_i(b_{i,k_i}) \) so that we finally reach the body atom \( b_{n,t} \) of \( r_n \) where \( Q_n = \theta_n(b_{n,t}) \) is the query which unifies with \( R(a_1, \ldots, a_n) \).

We hence obtain a sequence \( Q_0 \ldots Q_n \) satisfying items 1 and 2. We shall show that for every sequence satisfying items 1 and 2 there is a sequence \( Q_0’ \ldots Q_m’ \) satisfying items 1–3:

The claim is clear, if the sequence is of length 1, i.e. \( n = 0 : Q_0 \) is never blocked. Let \( Q_0 \ldots Q_n \) be a sequence of length \( n + 1 \) with \( Q_i \) equals \( Q_k \) up to variable renaming where \( 0 \leq i < k \leq n \). Then the head of \( r_{k+1} \) unifies with the query \( Q_i \). The sequence \( Q_0, \ldots, Q_i, Q_{k+1}, \ldots, Q_n \) is properly shorter than \( Q_0 \ldots Q_n \) and satisfies items 1–2. The induction hypothesis yields a sequence \( Q_0’, \ldots, Q_m’ \) which satisfies items 1–3.

**Proposition 2** Let \( Q \) be an input query for function main and \( R(a_1, \ldots, a_k) \) a fact, which appears in the proof-tree of some fact derived from \( Q \) under RuleSet in Database then there is a while-loop pass from which onwards \( R(a_1, \ldots, a_k) \) is returned by every query \( Q_n \), produced by \( \text{infer}(Q, Database, \emptyset) \), which unifies with \( R(a_1, \ldots, a_k) \).

**Proof 2** We prove by induction upon \( n < \omega \), that the fact \( R(a_1, \ldots, a_k) \) is yielded in at most \( n \) many while-loop passes, if the height of the minimal proof-tree for \( R(a_1, \ldots, a_k) \) is equal to \( n \).

Proposition 1 shows that there is an unblocked query \( Q_n \) produced by \( \text{infer}(Q, RuleSet, \emptyset) \) such that \( R(a_1, \ldots, a_k) \) unifies with \( Q_n \).

If the proof-tree is of height 0, then \( R(a_1, \ldots, a_k) \) is a fact in Database and \( \text{infer} \) will always produce this fact in the look-up of line 13 which will be returned (cf. line 27) by \( \text{infer} \) for all while-loop passes.

Assume the proof-tree is of height > 0. Then there is a rule \( r : R(\bar{t}) \leftarrow R_{i_1}(\bar{t}_1) \land \ldots \land R_{i_m}(\bar{t}_m) \) and a variable assignment \( \beta \) such that \( R(\beta(\bar{t})) = R(a_1, \ldots, a_k) \) and for each \( i \in \{1, \ldots, m\} \) the fact \( R_i(\beta(\bar{t}_i)) \) has a proof-tree of height at most \( n \) under RuleSet in Database.

The head \( h \) of \( r \) unifies with \( R(a_1, \ldots, a_k) \). Let \( \theta := \text{MGU}(Q_n, h) \) then each \( R_i(\theta(\bar{t}_i)) \) unifies with
4. Hybrid Reasoning: Pre-Materialization

Our approach relies on a pre-materialization phase before the user can query the knowledge base where we calculate some subqueries so that during query-time our backward-chaining algorithm is able to infer all the derivation. We first formalize and discuss the method of pre-materialization and we will then show that suppressing the evaluation of pre-materialized subqueries leads to the same query answers that can be inferred with the original Database and the original program. Our notation and notions will closely follow those used in [1] and there especially Chapter 12.

4.1. Pre-Materialization

Let \( \mathcal{J} \) be a database and \( P \) the program with a list \( L \) of atomic queries that are selected for pre-materialization. The pre-materialization is performed by Algorithm 2. The reason why we do not simply introduce auxiliary relations \( S_Q \) to \( \mathcal{J} \) for each \( Q \in L \) and populate these by setting \( S_Q := \text{main}(Q) \) (for \( \text{main} \) cf. Algorithm 1) is that the efficiency of Algorithm 1 hinges upon that as many body atoms as possible are unfoldable, but are edbs for which merely look-ups have to be performed during backward chaining. We shall now explain Algorithm 2 in detail and discuss its completeness: In a first step (lines 1–3), the database is extended with auxiliary relations named \( S_Q \) for \( Q \in L \). Each rule of the program \( P \) is rewritten (lines 5–12) by replacing every body atom \( R_i(t_i) \) with the query \( S_Q(t_i) \) if \( R_i(t_i) \subseteq Q \), i.e. if the “answers” to \( R_i(t_i) \) are also yielded by \( Q \). The new rule thus obtained is stored in a new program \( P' \). In case the rule \( p \) contains no body atoms that need to be replaced, \( p \) is stored in \( P' \) as well.

In each repeat-loop pass (cf. lines 15–24), \( \mathcal{J} \) is extended in an external step (lines 17–19) with all answers for \( Q \in L \), which are stored in the auxiliary relation \( S_Q \). Since this is repeated between each derivation until no new answers for any \( Q \in L \) are yielded, this is equivalent to adding \( S_Q(t_i) := R(t_i) \) for each \( R(t_i) \in \)

Q_i := R_i(\theta(t_i)). Since \( Q_n \) is not blocked, every \( Q_i \) is a subquery of \( Q_n \).

By the induction hypothesis, for all \( i \in \{1, \ldots, n\} \), every \( Q_i \) occurring in the computation yielded \( R_i(\beta(t_i)) \) after at most \( n \) many while-loop passes. Therefore \( R(a_1, \ldots, a_n) \) was returned by this particular \( Q_n \) at the very latest in the \( n \)-th while-loop pass and eventually added to \( Mat \) (cf. line 4). Hence in at most \( n + 1 \) while-loop passes, every subsequent query \( Q_n \) returns \( R(a_1, \ldots, a_n) \) as look-up in line 12.
with \( Q = R(\bar{t}) \) to \( P' \) directly. This makes the procedure complete in the sense, that after termination of this algorithm \( S_0^Q \) contains all answers for the query \( Q \) in the full materialization of Database under Ruleset.

**Example 4** Take the altered program from Example 1 and add the appropriate \( S_0^Q(Q) \) \( \leftarrow \) \( T(Q) \) to it. In this case we obtain

\[
T(a, p1, b) \leftarrow S(p, SPO, p1) \land T(a, p, b)
\]

\[
T(x, SPO, y) \leftarrow S(x, SPO, w) \land S(w, SPO, y)
\]

\[
S(x, SPO, y) \leftarrow T(x, SPO, y)
\]

It is trivially clear that this program yields for every Database exactly the same results for \( T(x, SPO, y) \) as the original program

\[
T(a, p1, b) \leftarrow T(p, SPO, p1) \land T(a, p, b)
\]

\[
T(x, SPO, y) \leftarrow T(x, SPO, w) \land T(w, SPO, y)
\]

Algorithm 2 terminates and is sound in the sense that after Algorithm 2 has terminated \( S_0^Q \) \( \supseteq Q^{P(\mathcal{J})} \) for all \( Q \in L \), i.e. every answer that could be derived from \( Q \) under \( P \) in \( \mathcal{J} \) is contained in \( S_0^Q \).

**Proposition 3** Algorithm 1 is complete in the sense that for the database \( \mathcal{J}_0 \) which we obtain after Algorithm 2 has terminated \( S_0^Q \) \( \supseteq Q^{P(\mathcal{J})} \) for all \( Q \in L \), i.e. every answer that could be derived from \( Q \) under \( P \) in \( \mathcal{J} \) is contained in \( S_0^Q \).

**Proof 3** Assume for the sake of contradiction Algorithm 1 were not complete: Assume no new element could be derived in line 22 from the current state of the database \( \mathcal{J}_0 \) but for some \( Q \in L \), \( \text{main}(Q) \) could derive another yet unknown fact from \( \mathcal{J}_0 \) using the original program \( P \). Let therefore \( R(a_1, \ldots, a_n) \) be the first yet undervield answer for any \( Q \in L \) which is derived under the original program \( P \).

Line 18 guarantees that all \( S_0^Q \) \( = Q^{\mathcal{J}_0} \) and so program \( P' \) is at this stage indistinguishable from \( P \). Hence \( \text{main}(Q) \) must derive the fact \( R(a_1, \ldots, a_n) \) under \( P' \), as well. A contradiction! Since \( S_0^Q \) never shrinks during the pre-materialization process, Algorithm 2 is complete.

### 4.2. Reasoning with Pre-Materialized Predicates

We shall now show that replacing body atoms with auxiliary predicates that contain the full materialization of the body atom w.r.t. a given database, yields the same full materialization of the database as under the original program. The claim will be shown in all its generality explaining on the way, how the theoretical setting we draw up is connected to our specific case.

Let \( P \) be an arbitrary Datalog program (in our specific case it would be the OWL RL ruleset) and \( \mathcal{J} \) a database. We assume that \( \mathcal{J} \) has already been enriched with the results of the pre-materialization.

As an example, assume the binary relation \( S^3 \) contains all answer tuples of the query

\[
\text{query}(x, y) \leftarrow T(x, SPO, y).
\]

under the program \( P \).

From an abstract point of view we can define \( S \) as an extensional database predicate (edb) of \( P \), i.e. it is not altered by \( P \) so that the interpretation \( S^3 \) of \( S \) under \( \mathcal{J} \) equals the interpretation \( S^{P(\mathcal{J})} \) of \( S \) under the least fix-point model \( P(\mathcal{J}) \) of \( P \) over \( \mathcal{J} \).

Since \( S \) is an edb and therefore does not appear in the head of any rule of \( P \), \( S \) cannot be unfolded and so the evaluation of \( S \) during the backward chaining process is reduced to a mere look-up in the database.
Example 5 Example 1 would then look like
\[ T(a, p1, b) \leftarrow S(p, p1) \land T(a, p, b) \]
\[ T(x, SPO, y) \leftarrow S(x, y) \]
and it is intuitively clear that this program will yield the same results for \( T \) as Example 1.

Such a replacement is harmless only if \( \mathcal{J} \) has been adequately enriched. Thus, the question arises which abstract conditions must be satisfied to allow such a replacement: In essence, we want that a rule fires under “almost the same” variable assignment as its replacement, which we formalize in the following two paragraphs.

Assume \( R_0, \ldots, R_n \) are predicates of our program \( P \). Let \( R(\vec{t}_0) \leftarrow R_1(\vec{t}_1) \land \ldots \land R_n(\vec{t}_n) \) be a rule in \( P \). The \( \vec{t}_i = (\vec{t}_{i,1}, \ldots, \vec{t}_{i,m_i}) \) represent tuples of terms, where each term is either a variable or an element of the domain in \( \mathcal{J} \) and \( m_i \) equals to the arity of \( R_i \) for all \( i \in \{1, \ldots, n\} \).

We define two queries, one being the body of the rule and one being the body of the rule where one body atom is replaced by \( S \): Let \( \vec{z} := \vec{t}_0 \cdot \vec{t}_1 \ldots \vec{t}_{i-1} \cdot \vec{t}_{i+1} \ldots \vec{t}_n \), i.e. the concatenation of all tuples except \( \vec{t}_i \) and let \( \vec{f} \) be some arbitrary tuple.

\[ q_0(\vec{z}) \leftarrow R_1(\vec{t}_1) \land \ldots \land R_i(\vec{t}_i) \land \ldots \land R_n(\vec{t}_n) \]
\[ q_1(\vec{z}) \leftarrow R_1(\vec{t}_1) \land \ldots \land S(\vec{f}) \land \ldots \land R_n(\vec{t}_n) \quad (\ast) \]

Now, the rule and its replacement fire under almost the same variable assignment if \( q_0^{P(\mathcal{J})} = q_1^{P(\mathcal{J})} \), i.e. \( q_0 \) and \( q_1 \) yield the same answers under \( P \) in \( \mathcal{J} \). We see, that it is “almost the same” variable assignment, as we do not require variable assignments to coincide on \( \vec{t}_i \) and \( \vec{f} \). In this way we do not require, e.g., \( S^3 = R_i^{P(\mathcal{J})} \). \( S \) is merely required to contain the necessary information. This is important, if we want to apply the substitution to rdf-triples, where we lack distinguished predicate names:

Example 6 Since there is only one generic predicate symbol \( T \), requiring \( S^3 = T^{P(\mathcal{J})} \) would mean that \( S \) contains the complete materialization of \( \mathcal{J} \) under \( P \) which would render our approach obsolete.

Also notice, that it is not sufficient to merely require \( q_0(\vec{t}_0)^{P(\mathcal{J})} = q_1(\vec{t}_0)^{P(\mathcal{J})} \), i.e. that both queries yield the same answer tuples \( \vec{t}_0 \) under \( P(\mathcal{J}) \), as the following example shows.

Example 7 Let the program \( P \) which computes the transitive closure of \( R_0 \) in \( R_1 \) consist of the two rules:
\[ R_1(x, z) \leftarrow R_1(x, y) \land R_0(y, z) \]
\[ R_1(x, y) \leftarrow R_0(x, y) \]
Consider database \( \mathcal{J} \) with \( R_0^\mathcal{J} := \{(a, b), (b, c)\} \). In the least fix-point model \( P(\mathcal{J}) \) of \( P \) we expect \( R_1^P(\mathcal{J}) = \{(a, b), (b, c), (a, c)\} \). Let \( S \) have the interpretation \( S^3 = \{(b, b), (c, c)\} \). Since \( R_1^P(\mathcal{J}) \) is the transitive closure, the following two queries deliver the same answer tuples under \( P(\mathcal{J}) \):
\[ q_0(x, z) \leftarrow R_1(x, y) \land R_0(y, z) \]
\[ q_1(x, z) \leftarrow R_1(x, y) \land S(y, z) \]

Yet the program \( P' \)
\[ R_1(x, z) \leftarrow R_1(x, y) \land S(y, z) \]
\[ R_1(x, y) \leftarrow R_0(x, y) \]

will not compute the transitive closure of \( R_0 \) in \( R_1 \), as \( R_1^{P'(\mathcal{J})} = \{(a, b), (b, c)\} \).

We shall now show that substituting a body atom \( R_i(\vec{t}_i) \) by \( S(\vec{f}) \) under the condition that the queries in \( (\ast) \) yield the same answer tuples under \( P(\mathcal{J}) \), generates the same least fix-point:

Proposition 4 Let \( P' \) be the program \( P \) where the rule \( R_0(\vec{t}_0) \leftarrow R_1(\vec{t}_1) \land \ldots \land R_i(\vec{t}_i) \land \ldots \land R_n(\vec{t}_n) \) in \( P \) has, for some tuple \( \vec{f} \) and db \( S \), been replaced by \( R_0(\vec{t}_0) \leftarrow R_1(\vec{t}_1) \land \ldots \land S(\vec{f}) \land \ldots \land R_n(\vec{t}_n) \).

Let \( q_0 \) and \( q_1 \) be defined as in \( (\ast) \).

If \( q_0^{P(\mathcal{J})} = q_1^{P(\mathcal{J})} \) then \( P(\mathcal{J}) = P'(\mathcal{J}) \).

Proof 4 Let \( T_P \) and \( T_{P'} \) be the appropriate fix-point operators (cf. [11], Chapter 13) for each program. We show for all \( k < \omega \) that if \( Q(a_1, \ldots, a_m) \in T_P^k(\mathcal{J}) \) then there is an \( \ell < \omega \) such that \( Q(a_1, \ldots, a_m) \in T_{P'}^\ell(\mathcal{J}) \) and vice versa. Since we start out from the same database \( \mathcal{J} \) we have \( T_P^0(\mathcal{J}) = T_{P'}^0(\mathcal{J}) \) which settles the base case.

Let \( R \) be an intensional predicate of \( P \), i.e. it appears in some rule head in \( P \). If \( R(a_1, \ldots, a_m) \in T_P^{k+1}(\mathcal{J}) \) then either \( R(a_1, \ldots, a_m) \in T_P^k(\mathcal{J}) \) and we are done or there is some rule \( R(\vec{t}_0) \leftarrow R_1(\vec{t}_1) \land \ldots \land R_n(\vec{t}_n) \) and some variable assignment \( \beta \) such that \( \beta(\vec{t}_0) = (a_1, \ldots, a_m) \) and \( R_j(\beta(\vec{t}_j)) \in T_P^\ell(\mathcal{J}) \) for all \( j \in \{1, \ldots, n\} \).

If \( R(\vec{t}_0) \leftarrow R_1(\vec{t}_1) \land \ldots \land R_n(\vec{t}_n) \in P' \), i.e. none of its body atoms where substituted, the induc-
tion hypothesis shows for each $j \in \{1, \ldots, n\}$ that we can find $t_j < \omega$ such that $R_j(\beta(t_j)) \in T_{P}^0(\mathcal{A})$. Let $t_0 := \max\{t_j \mid 1 \leq j \leq n\}$. Then $R_j(\beta(t_j)) \in T_{P}^0(\mathcal{A})$ for all $j \in \{1, \ldots, n\}$. Since all premises of this rule are satisfied, there is some $\ell < \omega$ such that $R_0(\beta(t_0)) \in T_{P}^0(\mathcal{A})$.

If $R(t_0) \leftarrow R_1(t_1) \wedge \ldots \wedge R_n(t_n) \notin P'$ it is the rule where $R_i(t_i)$ has been substituted with $S(t)$. For the assignment $\beta$ we now know $\beta(t_0 \cdot t_1 \cdot \ldots \cdot t_{i-1} \cdot \bar{t}_{i+1} \cdot \ldots \cdot \bar{t}_n) \in q_0^P(\mathcal{A})$. Since $q_0^P(\mathcal{A}) = q_1^P(\mathcal{A})$ we know that there is some assignment $\beta'$, which coincides with $\beta$ on $(t_0 \cdot t_1 \cdot \ldots \cdot t_{i-1} \cdot \bar{t}_{i+1} \cdot \ldots \cdot \bar{t}_n)$ such that $\beta'(\bar{t}_i) \in S(P^0(\mathcal{A}))$.

Hence $R_j(\beta'(t_j)) \in T_{P}^0(\mathcal{A})$ for all $j \in \{1, \ldots, n\} \setminus \{i\}$ and $S(\beta'(t_i)) \in T_{P}^0(\mathcal{A})$ since $S$ is an edb predicate. The induction hypothesis yields some $t_j < \omega$ for each $j \in \{1, \ldots, n\} \setminus \{i\}$ such that $R_j(\beta'(t_j)) \in T_{P}^0(\mathcal{A})$. Let $t_0 := \max\{t_j \mid 1 \leq j \leq n$ and $j \neq i\}$. Then $R_j(\beta'(t_j)) \in T_{P}^0(\mathcal{A})$ for all $j \in \{1, \ldots, n\}$ and $S(\beta'(t_i)) \in T_{P}^0(\mathcal{A})$. Since all premises of this rule are satisfied, there is some $\ell < \omega$ such that $R_0(\beta'(t_0)) \in T_{P}^0(\mathcal{A})$. As $\beta$ coincides with $\beta'$ also on $t_0$, i.e. $\beta'(t_0) = (a_0, \ldots, a_m)$, we have in particular $R(a_0, \ldots, a_m) \in T_{P}^0(\mathcal{A})$.

This shows that for all predicates $Q$ we have $Q^P(\mathcal{A}) \subseteq Q^{P'}(\mathcal{A})$. For the converse we merely show the case of the substituted rule: Assume $R(a_1, \ldots, a_m) \in T_{P+1}^0(\mathcal{A})$ and there is a assignment $\beta'$ such that $\beta'(t_0) = (a_1, \ldots, a_m)$ and $R_j(\beta'(t_j)) \in T_{P}^0(\mathcal{A})$ for all $j \in \{1, \ldots, n\} \setminus \{i\}$ as well as $\beta'(t_i) \in S(P^0(\mathcal{A}))$.

The induction hypothesis yields for each $j \in \{1, \ldots, n\} \setminus \{i\}$ some $t_j < \omega$ with $R_j(\beta'(t_j)) \in T_{P}^0(\mathcal{A})$. Since $S$ is an edb predicate for $P$, we have $S(\beta'(t_i)) \in T_{P}^0(\mathcal{A})$. Hence for $t_0 := \max\{t_j \mid 1 \leq j \leq n$ and $j \neq i\}$ we have $R_j(\beta'(t_j)) \in T_{P}^0(\mathcal{A})$ and $S(\beta'(t_i)) \in T_{P}^0(\mathcal{A})$.

This implies $\beta'(t_0 \cdot t_1 \cdot \ldots \cdot t_{i-1} \cdot \bar{t}_{i+1} \cdot \ldots \cdot \bar{t}_n) \in q_1^P(\mathcal{A})$ and since $q_0^P(\mathcal{A}) = q_1^P(\mathcal{A})$ there is a assignment $\beta$ coinciding on $(t_0 \cdot t_1 \cdot \ldots \cdot t_{i-1} \cdot \bar{t}_{i+1} \cdot \ldots \cdot \bar{t}_n)$ with $\beta'$ such that $R_i(\beta(t_i)) \in T_{P}^0(\mathcal{A})$ for some $j_0 < \omega$. Let $t_1 := \max\{t_0, j_0\}$ then $R_i(\beta(t_i)) \in T_{P}^0(\mathcal{A})$ for all $j \in \{1, \ldots, n\}$. Since all premises of the rule $R_0(t_0) \leftarrow R_1(t_1) \wedge \ldots \wedge R_n(t_n)$ are satisfied, there is some $\ell < \omega$ such that $R_0(\beta(t_0)) \in T_{P}^0(\mathcal{A})$, which shows, as $\beta$ coincides on $t_0$ with $\beta'$ that $R(a_0, \ldots, a_m) \in T_{P}^0(\mathcal{A})$.

Together with $Q^P(\mathcal{A}) \subseteq Q^{P'}(\mathcal{A})$ this shows $Q^P(\mathcal{A}) = Q^{P'}(\mathcal{A})$ for all predicate names $Q$ and hence that $P(\mathcal{A}) = P'(\mathcal{A})$.

In the following section we shall apply this rewriting on the OWL RL ruleset.

5. Hybrid reasoning for OWL RL

In the previous sections we described the two main components of our method which consists of the backward-chaining algorithm used to retrieve the inference and the pre-materialization procedure which ensures the correctness of our approach.

We will now discuss the implementation of the OWL RL rules using our approach.

The official OWL RL ruleset contains 78 rules, for which the reader is referred to the official document overview [11]. With some selected examples from [11] we will illustrate some key features of our algorithm.

Since our intention is to derive new triples, we exclude from our discussion those rules whose purpose is to derive an inconsistency, i.e. rules with predicate false in the head of the rule. We also ignore rules which have an empty body because they cannot be triggered during the unfolding process of backward-chaining. These rules can be trivially implemented by either adding a fixed number of axiomatic triples or with a single pass on the data.

The rules that exploit the owl:sameAs transitivity and symmetry require a computation which is too expensive to perform at query time since they can be virtually applied to every single term of the triples. We tackled this issue by precomputing beforehand the owl:sameAs closure and consolidation which is a common practice among reasoners. This procedure is explained in detail at [16].

In our approach we decided to pre-materialize all triple patterns that are used to retrieve “schema” triples, also referred to as the terminological triples. In Table 6 of Appendix A we report the complete list of the patterns that are pre-materialized using our method described in Section 4.

Singling out exactly those triple patterns from Table 6 is motivated by the empirical observation that:

- they appear in many of the OWL rules;
- their answer sets are very small compared to the entire input;
- their answer sets are not as frequently updated as the rest of the data.

\[2\] The list of these rules is reported in Table 4 of the document at [11].
These characteristics make the set of inferred schema triples the ideal candidate to be pre-materialized. Along with them, at every step of the pre-materialization, we also query the database to retrieve all the triples with the rdf:first, rdf:next and owl:sameAs predicates. The first two are necessary to calculate the lists that should be used to execute the rules with a dynamic number of antecedents. The third is used to check whether the sameAs consolidation table should be updated with new information.

All rules which have a pre-materialized pattern amongst their body atoms, are substituted replacing the pre-materialized pattern with its corresponding auxiliary relation as justified by Proposition 4. This affects 25 rules out of 78 and hence reduces reasoning considerably.

After the pre-materialization procedure is completed, each rule which has a pre-materialized pattern in its head can be reduced to a mere look-up:

**Example 8** Consider $(scm$-sco) from Table 9 in [11]:

$T(x, SCO, z) \leftarrow T(x, SCO, y) \land T(y, SCO, z)
$

*can be replaced by Proposition 4*

$T(x, SCO, z) \leftarrow S_{sco}(x, SCO, y) \land S_{sco}(x, SCO, z)$

where all answers to $T(?c_1, SCO, ?c_2)$ are contained in $S_{sco}^3$. Then $T(x, SCO, y)$ can be replaced by $\forall$ the constant for true, again using Proposition 4, and finally obtaining

$T(x, SCO, z) \leftarrow S_{sco}(x, SCO, z)$

This removes a further 30 rules from unfolding.

Since the OWL RL fragment consists of a large number of rules, there is an high possibility that the proof tree contains branches that lead to the same derivation. Detecting an avoiding the execution of these branches is essential in order to reduce the computation.

After empirical analysis on some example queries we determined that there are two types of sources in the generation of duplicates. The first comes from the nature of the ruleset. The second comes from the input data.

**First type of duplicates source.** The most prominent example of generation of duplicates of the first type is represented by the symmetric rules which have the same structure but have the variables positioned at different locations. We refer with rule names and names in the following list to the OWL RL ruleset in [11]:

prp-eqp1 and prp-eqp2 from Table 5
as cax-eqc1 and cax-eqc2 from Table 7
prp-inv1 and prp-inv2 from Table 5.

We will now analyze each of these three cases below.

Let $S_{eqp}^2$ be the pre-materialization of the triple pattern $T(x, EQP, y)$. Rules $scm$-eqp1 and $scm$-eqp2 render $S_{eqp}^2$ symmetric. Hence

$q(x, p_2, y) \leftarrow T(x, p_1, y) \land S_{eqp}(p_1, EQP, p_2)$

yields the same results under $P(\exists)$ as

$q(x, p_1, y) \leftarrow T(x, p_2, y) \land S_{eqp}(p_2, EQP, p_1)$

and Proposition 4 yields that $scm$-eqp2 can be replaced by $scm$-eqp1, effectively deleting $scm$-eqp2 from the ruleset.

Similarily, rules $scm$-eqc1 and $scm$-eqc2 render the results of the pre-materialized query $T(x, EQC, y)$, symmetric.

In contrast, the pre-materialized query $T(x, INV, y)$ is not symmetric. However, we can first observe that Proposition 4 allows to replace the rules by

$T(x, p, y) \leftarrow T(x, q, y) \land S_{inv}(p, INV, q)$

$T(x, p, y) \leftarrow T(x, q, y) \land S_{inv}(q, INV, p)$

which defuses the idb atom $T(q, INV, p)$ into the harmless edb $S_{inv}$, for which $S_{inv}^2$ contains all answers to the query $T(x, INV, y)$. Let this new program be called $P''$. Further, let $P'''$ be the program where both rules have been replaced by

$T(x, p, y) \leftarrow T(x, q, y) \land S_{inv}^2(p, INV, q)$

with $S_{inv}^2$ being the symmetric closure of $S_{inv}^3$. It is now not difficult to see, that every model of $P'''$ is a model of $P''$ and vice versa. In particular the least fix-point model of $P(\exists)$ is equal to the least fix-point model $P''(\exists)$. Hence we can replace prp-inv1 and prp-inv2 by one rule under the condition that we pre-materialize the symmetric closure of $T(x, INV, y)$.

**Second type of duplicates source.** The second type of duplicate generations comes from the input data which might contain some triples that make the application of two different rules perfectly equivalent.

We have identified an example of such a case in the Linked Life Dataset, that is one realistic dataset that we used to evaluate our approach. In this dataset there is the triple:

$T(SCO, TYPE, TRANS)$

which states that the subClassOf predicate is transitive.

In this case, during the precomputation phase the query $T(a, SCO, b)$ will be launched several times, and
each time the reasoner will trigger the application of both the rules \textit{scm-sco} and \textit{prp-trp}.

However, since the application of these two rules will lead to the same derivation, such computation is redundant and inefficient. Therefore, to detect such cases we apply a special algorithm (which will not be presented here) at loading time to calculate whether some rules should not be applied in certain cases to prevent the derivation of duplicate answers.

6. Evaluation

We have implemented our approach in a Java prototype that we called \textit{QueryPIE} \(^3\) and we evaluated the performance using one machine of the DAS-4 cluster\(^4\), which is equipped with a dual Intel E5620 quad core CPU of 2.4 GHz, 24 GB of memory and 2 hard disks of 1 TB each.

We used two datasets as input. LUBM \([7]\) which is one of the most popular benchmarks for OWL reasoning and LLD (Linked Life Data)\(^5\), which is a curated collection of real-world datasets in the bioinformatics domain.

LUBM allows to generate datasets of different sizes. For our experiments we generated a dataset of 10 billion triples (which corresponds to the generation of 80000 LUBM universities). The Linked Life Data dataset consists of about 5 billion triples.

Both datasets were compressed using the procedure described in \([17]\). The QueryPIE prototype uses six indices stored alongside with the triple permutations on disk using an optimized B-Tree data structure.

We organized this section as follows. First, in Section 6.1 we will report a set of experiments to evaluate the performance of the pre-materialization phase. Next, in Section 6.2 we will focus on the performance of the backward-chaining approach and analyze its performance on some example queries.

6.1. Performance of the pre-materialization algorithm

We launched the pre-materialization algorithm on the two datasets to measure the reasoning time necessary to perform the partial closure. The results are reported in the second column of Table 1. Our prototype performs joins between the pre-materialized patterns when it loads the rules in memory, therefore, we have also included the startup time along with the query runtimes to provide a fair estimate of the time requested for the reasoning.

From the results, we notice that the pre-materialization is about two orders of magnitude faster for the LUBM dataset than for LLD. The reason behind this difference is that the ontology of LUBM requires much less reasoning than the one of LLD in order to be pre-materialized. In fact, in the first case the pre-materialization algorithm has derived 390 triples and needing four iterations to reach a fix point. On the other case the pre-materialization required 7 iterations and returned about 10 million triples.

We intended to compare the cost of performing the partial closure against a the cost of a full materialization, which is currently considered as the state of the art in the field of large scale OWL reasoning. However, to the best of our knowledge there is no approach described in literature which supports the OWL RL fragment and which is able to scale to the input size that we consider.

The closest approach we can use for a comparison is WebPIE \([16]\), which has demonstrated OWL reasoning up to the \(pD^*\) fragment to a hundred billion triples. Since WebPIE uses the MapReduce programming model, an execution on a single machine would be suboptimal. Therefore, we launched it using eight machines and multiplied the execution time accordingly to estimate the runtime on one machine (such estimation is in line with the performance of WebPIE which has shown linear scalability in \([16]\)).

The runtime of the complete materialization performed with this method is reported in the third column of Table 1. We notice that in both cases a complete materialization requires between four and five days against the seconds or minutes required for our method. This comparison clearly illustrates the advan-

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Dataset & Reasoning time & N. iterations & N. derived triples \\
 & Our approach & Full materialization & & \\
\hline
LUBM & 6.0s & 4d1h16m & 4 & 390 \\
LLD & 911.4s & 5d10h45m & 7 & 10 millions \\
\hline
\end{tabular}
\caption{Execution time of the pre-materialization algorithm compared to a full closure.}
\end{table}

\(^3\)We intend to release all the software with an opensource license and plan to reference it in case this paper will be accepted.

\(^4\)http://www.cs.vu.nl/das4

\(^5\)http://www.linkedlifedata.com/
Urbani et al. / QueryPIE: Hybrid Reasoning With The OWL RL Rules

Table 2
List of example queries

<table>
<thead>
<tr>
<th>ID</th>
<th>Dataset</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LUBM</td>
<td>?x ?y <a href="http://www.Department0.University0.edu/GraduateCourse0">http://www.Department0.University0.edu/GraduateCourse0</a></td>
</tr>
<tr>
<td>2</td>
<td>LUBM</td>
<td>?x <a href="">lubm:subOrganizationOf</a> <a href="http://www.University0.edu">http://www.University0.edu</a></td>
</tr>
<tr>
<td>3</td>
<td>LUBM</td>
<td><a href="http://.../GraduateStudent124">http://.../GraduateStudent124</a> <a href="">lubm:degreeFrom</a> <a href="http://www.University114.edu">http://www.University114.edu</a></td>
</tr>
<tr>
<td>4</td>
<td>LUBM</td>
<td>?x ?y <a href="http://www.Department0.University0.edu/AssistantProfessor0">http://www.Department0.University0.edu/AssistantProfessor0</a></td>
</tr>
<tr>
<td>5</td>
<td>LUBM</td>
<td>?x <a href="">lubm:memberOf</a> <a href="http://www.Department0.University0.edu">http://www.Department0.University0.edu</a></td>
</tr>
<tr>
<td>6</td>
<td>LUBM</td>
<td>?x <a href="">rdf:type</a> <a href="">lubm:Department</a></td>
</tr>
<tr>
<td>7</td>
<td>LLD</td>
<td>?x ?y <a href="">lifeskim:mentions</a></td>
</tr>
<tr>
<td>9</td>
<td>LLD</td>
<td><a href="http://.../resource/pubmed/id/15964627">http://.../resource/pubmed/id/15964627</a> ?x ?y</td>
</tr>
<tr>
<td>10</td>
<td>LLD</td>
<td>?x ?y <a href="http://purl.uniprot.org/go/0006952">http://purl.uniprot.org/go/0006952</a></td>
</tr>
</tbody>
</table>

Table 3
Runtime of the queries in Table 2 on the LUBM and LLD datasets

<table>
<thead>
<tr>
<th>Q.</th>
<th>Runtime (ms)</th>
<th>Derived Triples</th>
<th>I/O access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cold</td>
<td>Warm</td>
<td>Total</td>
</tr>
<tr>
<td>1</td>
<td>163.45</td>
<td>11.9</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1207.22</td>
<td>145.51</td>
<td>941</td>
</tr>
<tr>
<td>3</td>
<td>129.39</td>
<td>7.2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>177.5</td>
<td>13.57</td>
<td>206</td>
</tr>
<tr>
<td>5</td>
<td>233.57</td>
<td>18.86</td>
<td>3518</td>
</tr>
<tr>
<td>6</td>
<td>6974</td>
<td>6641.69</td>
<td>3199974</td>
</tr>
<tr>
<td>7</td>
<td>382.91</td>
<td>31.64</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>350.4</td>
<td>20.53</td>
<td>8932</td>
</tr>
<tr>
<td>9</td>
<td>7499.29</td>
<td>678.16</td>
<td>823</td>
</tr>
<tr>
<td>10</td>
<td>9289.88</td>
<td>1227.38</td>
<td>58103</td>
</tr>
<tr>
<td>11</td>
<td>1571.91</td>
<td>106</td>
<td>9278</td>
</tr>
<tr>
<td>12</td>
<td>6299.49</td>
<td>5405.67</td>
<td>2375888</td>
</tr>
</tbody>
</table>

6.2. Performance of the reasoning at query time

In order to analyze the performance of reasoning at query time, we launched some example queries after we computed the closure using our backward-chaining algorithm to retrieve the results. To this purpose, we selected six example queries for both the LUBM and LLD datasets and report them in Table 2.

While LUBM provides an official set of queries for benchmarking, unfortunately there is no official set of queries that can be used for benchmarking the performance on the LLD dataset. Therefore, we took some queries that are reported in the official page of the LLD dataset and modified them so that they could trigger different types of reasoning.

These queries were selected according to the following criteria:

– Number of results: We selected queries that return a number of results that varies from no results to a large set of triples;

– Reasoning complexity: Some queries in our example set require no reasoning to be answered, in contrast other queries generate a very large proof-tree;

– Amount of data processed: In order to answer a query, the system might need to access and process a large set of data. We selected queries that read and process a variable amount of data to verify the impact of I/O on the overall performance.

We performed a number of experiments to analyze three aspects of the performance of our algorithm during query time: the absolute response time, the reduction of the proof-tree, and the overhead induced by reasoning during query-time. Each of these aspects is analyzed below.

6.2.1. Absolute response time

We report in Table 3 the execution time obtained launching the selected example queries in Table 2. In
the second and third columns we report both the cold and warm runtime. With cold runtime we identify the runtime that is obtained by launching the query right after the system has started. Since the data is stored on disk, with the cold runtime we also measure the time to read the data from disk. On the other side, the warm runtime measures the average response time of launching the same query ten more times. Because during such execution the data is already cached in memory and the Java VM has already initialized the internal data structures, the warm runtime is significantly faster than the cold one.

The fourth and fifth column, respectively, report the total number of derivations that were inferred during the execution of the query, and the number of triples returned to the user.

The sixth and seventh column report the number of data lookups required to answer the query and the amount of data that is read from disk. These two numbers are important to estimate the impact of reasoning at query time. While one query without reasoning requires only one data lookup, in our case the reasoning algorithm might require to access the database multiple times. For example, in order to answer query 10 the program had access to the data indices about 10000 times.

From the results reported in Table 3 we can make few considerations. First, we notice that the cold runtime is significantly slower than the warm runtime between one and two orders of magnitude. This is primarily due to expensive cost of the I/O access to disk especially because reasoning requires to read at different location of the data indices, and therefore the system is required to read several blocks of the B-Tree from the disk. For most of the queries, the I/O access dominates the execution time. The worst case is represented by query 10 where the program reads from disk about 382 MB of data. From these results we conclude that the performance of the program in case the data is stored on disk is essentially I/O bounded. After the data is loaded in memory, the execution time drops by about one order of magnitude on average and the performance becomes CPU bounded.

Another factor that impacts the performance is the number of the inferred triples that are calculated during the execution of the query. In fact, we notice that absolute performance is lower in case a large number of triples is either inferred or retrieved from the database. The behavior is due to the fact that the algorithm needs to temporarily store these triples as it must consider them in each repeat-loop pass until the closure is reached. This means that these triples must be stored and indexed to be retrieved during the following iterations and the response time consequently increases.

Summarizing our analysis, we make the following conclusions: (i) the runtime is influenced by several factors among which the most prominent is the amount of I/O access that is requested to answer the query (this number is proportional to the size of the proof tree) and the number of derivations produced. (ii) Even in the worst case where all the data is stored on disk and a large number of results must be retrieved, the absolute runtime can be still seen as interactive being in the range of few seconds. In the average and best cases, the performance is in the order of dozens of milliseconds which is in the same range of the performance of current high performance triple stores, yet with the advantage that we additionally enrich query results with implicit information inferred under the ruleset from the database.

### 6.2.2. Reduction of the proof tree

The backward-chaining algorithm and more in general our approach relies on the pre-materialization of some selected queries which serve a variety of purposes like performing efficient sideways information passing or excluding rules that derive duplicates. Another advantage of performing the pre-materialization is that it reduces the size of the proof tree during query-time.

In this section, we will evaluate the effective reduction in terms of the size of the proof tree obtained by avoiding performing inference on the pre-materialized patterns.

However, since the method presented in this paper is embedded in the implementation of our prototype, and since the optimizations introduced are crucial to its execution, we cannot disable them. To overcome this problem, we have manually analyzed the ex-
execution of the LUBM queries with our prototype on
a much smaller dataset and manually constructed the
proof tree without pre-materialization (note that we ex-
cluded queries 4 and 6 since in these cases reasoning
did not contribute to derive new answers). In prin-ci-
ple, for each query, we identified the rules that produce
some derivations and for each pre-materialized query
in their body, we added the corresponding branch that
was generated when that query was calculated during
the pre-materialization phase.

We report the results of such analysis in Table 4.
The last column reports the obtained reduction ratio
and shows that the number of leaves shrinks between
two and four times due to our pre-materialization.

The results of this method of evaluation must be
seen as an underestimate, because we could not de-
activate all the optimizations, and therefore in real-
ity the gain is even higher than the one calculated.
Nevertheless, this shows that our pre-calculation is
indeed effective. For a very small cost in both data space
and upfront computation time, we substantially reduce
the proof-tree. Apparently, the pre-materialization pre-
cisely captures small amounts of inferences that con-
tribute substantially to the reasoning costs because
they are being used very often.

6.2.3. Overhead of reasoning during query-time

While we are able to significantly reduce the size
of the proof-tree and apply other optimizations to fur-
ther reduce the computation, we still have to perform
some reasoning during the execution of a query. It is
important to evaluate what the cost for the remain-
ing reasoning is when we compare our approach to a
full-materialization approach (which is currently the
de-facto technique for large scale reasoning), where a
large pre-materialization is performed so that during
query time reasoning is avoided altogether.

To this end, we launched a number of experiments
activating different types of reasoning at query time
and report the results in Table 5.

We proceeded as follows: we first launch the queries,
deactivating all rules at query time, and state their ex-
ecution time in the first column of the table (the title
“No Ins.” indicates no insertion). We then reissued the
queries activating only the RDFS rules (in the third
column), then the $pD^*$ rules and finally the OWL RL
ones. Since the number of results is different due to the
different considered ruleset, we have repeated the same
experiments this time adding all the inferred triples to
the database. The response times are reported under
the column “Ins.”.

From the results presented in the table, we notice
that the response time progressively increases as we in-
clude more rules. Such a behavior is clearly expected
since more computation must be performed as we add
new rules. However, in some cases (like query 12)
there is a significant difference even if the query does
not require the application of any rule. The difference
is due to the cost of temporary storing the results to en-
sure the completeness of the backward-chaining algo-
rithm. This operation is clearly a non-negligible con-
tributor to the overall performance.

We can compare the response times reported in the
second column with the ones of the penultimate col-
umn to compare the performance of the reasoning at
query time of our approach against traditional full materialization. In fact, because the input data already contains the whole derivation, a single lookup can be used to estimate the cost that we would have to pay if all the inferences were pre-materialized beforehand. From the results we notice that on average the response time is between one and two order of magnitude slower. In case the query needs to process and/or return many triples, the difference is certainly significant. However, on average the response time is still in the order of the dozens of milliseconds and therefore, from the user perspective, the difference is much less noticeable.

7. Related Work

Applying rules with a top-down method like backward-chaining is a well-known technique in rule-based languages like Datalog [4]. In this work, we optimized the computation to exploit the characteristics of RDF data and execute a standard set of rules. Our backward-chaining algorithm is inspired by the QSQ algorithm and the traditional semi-naive evaluation algorithm which are well-known techniques in logic programming. A similar termination condition to ours is employed also in the RQA/FQI algorithm [10].

In our approach, we exploit the availability of the precomputation using a **sideway information passing (SIP)** technique during the execution of the rules. This technique is used in other approaches like in the magic set rewriting algorithm [2]. However, while the magic set algorithm uses it at compile-time to construct rules bottom-up, we employ this technique at runtime to execute queries in a top-down manner. Also, SIP strategies are similarly used in generic query processing to prune irrelevant results. In [8] the authors propose two adaptive SIP strategies where information is passed adaptively between operators that are executed in parallel.

Some RDF Stores support various types of inference. 4store [14] applies the RDF5 rules with backward-chaining. Virtuoso [6] supports the execution of few (but not all) OWL rules. BigOWLIM [3] is a RDF store that supports the OWL 2 RL ruleset by performing a full materialization when the data is being loaded. Another database system that performs OWL RL reasoning in a similar way is Oracle: In [9] the authors describe their approach reporting the performance of the inference over up to seven billion triples. Another approach in which the OWL RL rules are used is presented in [15] where the authors have encoded OWL RL reasoning in the context of embedded devices, and therefore optimizing the computation for devices with limited resources.

Some work has been presented to distribute the reasoning process using supercomputers or clusters of machines. In our previous work we used the MapReduce programming model to improve the scalability [16]. In [21], the authors implement RDFS reasoning using the BlueGene supercomputer. To the best of our knowledge none of these approaches supports the OWL RL rules.

Implicit information can be derived not only with rule-based techniques. In [12], the authors focus on ontology based query answering using the OWL 2 QL profile [11] and present a series of techniques based on query rewriting to improve the performance. While we demonstrate inference over a much larger scale, a direct comparison of our technique with this work is difficult since both the language and reasoning techniques are substantially different.

A series of work has been done on reasoning using the OWL EL profile. This language is targeted to domains in which there are ontologies with a very large number of properties and/or classes. [5] presented an extensive survey of the performance of OWL EL reasoners analyzing tasks like classification or consistency checking. Again, the different reasoning tasks and considered language makes a direct comparison difficult for our approach.

8. Conclusions

Until now, all inference engines that can handle reasonably expressive logics over very large triple stores (in the orders of billion of triples) have deployed full materialization. In the current paper we have broken with this mold, showing that it is indeed possible to do efficient backward-chaining over large and reasonably expressive knowledge bases.

The key to our approach is to precompute a small number of inferences which appear very frequently in the proof-tree. This of course re-introduces some amount of preprocessing, but this computation is measured in terms of minutes, instead of the hours needed for the full closure computation.

By pre-materializing part of the inference upfront instead of during query-time, we are able to introduce a number of optimizations that exploit such precomputation to improve the performance during query-time.
To this end, we adapted a standard backward-chaining algorithm like QSQ to our usecase exploiting the parallelization of current architectures.

Since our approach deviates from standard practice in the field, we have formalized the computation using the theory of deductive databases and extensively analyzed and proved its correctness.

We have implemented our method in a proof-of-concept Java prototype and analyzed the performance over both real and artificial datasets of five and ten billion triples using the OWL RL rules. The performance analysis shows that the query response-time for our approach is competitive with that of full materialization, with response times in the low number of milliseconds in the best cases, and increasing up to few seconds as the query increases in its complexity. The loss of response time is offset by the great gain in not having to perform a very expensive computation of many hours before being able to answer the first query.

Obvious next steps in future work would be to investigate how our approach can further scale in terms of data size and reasoning complexity and to understand the properties of the knowledge base that influence both the cost of the limited forward computation and the size of the inference tree. Also, it is worth to explore whether related techniques as ad-hoc query-rewriting like the one presented in [12] can be exploited to further improve the performance.

To the best of our knowledge, this is the first time that complete backward-chaining reasoning over realistic OWL RL knowledge bases of a ten billion triples has been realized. Our results show that this approach is feasible, opening the door to reasoning over much more dynamically changing datasets than was possible until now.

Acknowledgments: We would like to thank Stefan Schlobach, Barry Bishop, Peter Patel-Schneider, Boris Motik, and Ian Horrocks for providing useful comments and advice. We also would like to thank Ceriel Jacobs for the support in developing the program. This work was partly supported by the LarKC project (EU FP7-215535) and by the COMMIT project as well as the SEALS Project.

References


Appendix

A. List of tables

Table 6

<table>
<thead>
<tr>
<th>OWL RL</th>
<th>RDF/S</th>
<th>pD*</th>
</tr>
</thead>
</table>

Table 7

List of abbreviations for common URIs used in this paper.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full text</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>rdf:type</td>
</tr>
<tr>
<td>SCO</td>
<td>rdfs:subClassOf</td>
</tr>
<tr>
<td>SPO</td>
<td>rdfs:subPropertyOf</td>
</tr>
<tr>
<td>EQC</td>
<td>owl:equivalentClass</td>
</tr>
<tr>
<td>EQP</td>
<td>owl:equivalentProperty</td>
</tr>
<tr>
<td>INV</td>
<td>owl:inverseOf</td>
</tr>
<tr>
<td>SYM</td>
<td>owl:SymmetricProperty</td>
</tr>
<tr>
<td>TRANS</td>
<td>owl:TransitiveProperty</td>
</tr>
</tbody>
</table>

Table 6

Triple patterns that are precalculated considering the OWL RL rules.