QueryPIE: Hybrid Reasoning With The OWL RL Rules

Jacopo Urbani\textsuperscript{a,}\textsuperscript{*}, Robert Piro\textsuperscript{b} Frank van Harmelen\textsuperscript{a} Henri Bal\textsuperscript{a}

\textsuperscript{a} Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
Email: \{jacopo,frankh,bal\}@cs.vu.nl
\textsuperscript{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 an 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 \cite{16,9,21} and demonstrated reasoning upon several billion of triples. These methods apply the rules in a forward-chaining fashion, so that all the possible derivations are produced and stored with the original input. While these methods exhibit good scalability because they can efficiently exploit computational parallelism, 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 complete 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 before query time. Our method relies on a backward-chaining algorithm to calculate the inference. The backward-chaining algorithm exploits the fact that a partial materialization of some selected queries is available and the parallel computing power of modern architectures to reduce the computation.

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 is not dependent on a particular rule set, but holds for a generic rule set that can be expressed in Datalog.

For the evaluation, however, we will apply our method to the OWL RL rule set, which is the latest standard OWL profile designed to work 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 below one 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 one that supports the OWL RL rules, which are officially standardized by W3C. Also, this paper provides a theoretical analysis of the approach proving its correctness w.r.t. the considered rule set and presents an improved explanation and evaluation over larger datasets.

The remainder of this paper is organized as follows: in Section 2 we present at high level the main idea behind hybrid reasoning. The purpose of this section is to introduce the reader to our problem and to provide a high level overview of our approach.

Next, in Section 3, we will describe the backward-chaining algorithm that is used within our method to calculate the inference. Section 4 formalizes the pre-computation algorithm of hybrid reasoning and proves its correctness. Next, in Section 5, we focus on the execution of the OWL 2 RL/RDF rule set (for simplicity we will refer to it as the OWL RL rule set) 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 rule set: One is to compute the complete extension of a database under some given rule set before query time and the other is to infer only the necessary entries needed to yield a complete answer from the rule set 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, then a complete materialisation before query time 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 in between: 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 rule set 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, then 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.

**Example 1** Consider the two following rules from the OWL RL rule set:

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

where we use from now on the abbreviation \(SPO\) for reasons of space\(^1\) 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. If we use Datalog to implement these rules in a program, then we can replace each atom by some new atom, using an extensional database predicate \((edb)\)\(^2\), say \(S\). 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 inferable under the rule set 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 for an efficient execution. After this, we will formally discuss the correctness of our method.

### 3. Hybrid Reasoning: Backward-chaining

In the current and following sections, we will use the notation and notions that come from the Datalog theory, and more in particular from [1, Chapter 12], to formalize and to prove the correctness of our method.

To ease the understanding of our explanation, we will briefly recall some well-known notions of Datalog that will be frequently used. To the same purpose, we will also enrich our explanation with brief examples in order to facilitate the comprehension in case the reader is not completely familiar with the concepts that are being used.

---

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

\(^2\)In Datalog, \(ed\) predicates are predicates that do not appear in the head of any rule. Therefore, they can be implemented much more efficiently since they require only a single lookup in the database.
Let \( \mathcal{I} \) be a generic Datalog database and \( R \) be a predicate symbol of arity \( n \). We denote with \( R^3 \) the \( n \)-ary relation named \( R \) in \( \mathcal{I} \). In a similar fashion we denote with \( q(\bar{x})^3 \) the set of all answers to a Datalog query \( q(\bar{x}) \) in \( \mathcal{I} \).

In our formalization, we denote \( T_\mathcal{I} \) as the immediate consequence operator of the Datalog program \( P \). An immediate consequence operator is an operator that maps a database \( \mathcal{I} \) to the database \( T_P(\mathcal{I}) \), where \( T_P(\mathcal{I}) \) is \( \mathcal{I} \) extended by all facts that could be inferred from facts in \( \mathcal{I} \) under \( P \). We define \( T_P^0(\mathcal{I}) := \mathcal{I} \) and \( T_P^{n+1} = T_P \circ T_P^n \).

With \( \omega \) we indicate the first infinite limit ordinal and set \( T_P^\omega(\mathcal{I}) := \bigcup_{n<\omega} T_P^n(\mathcal{I}) \). According to [1, Chapter 12], we have \( P(\mathcal{I}) = T_P^\omega(\mathcal{I}) \) and in particular that for every fact \( \bar{a} \in R^P(\mathcal{I}) \) there must be some \( n < \omega \) such that \( \bar{a} \in R^{T_P^n}(\mathcal{I}) \).

We will now discuss our backward-chaining algorithm. The purpose of this algorithm is to derive all possible triples that are part of a given input query \( Q \), given a database \( D \) and a rule set \( R \).

Conventionally, 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. Since RDF contains only one generic predicate symbol \( T \), it is not mentioned in the query.

The inference rules in the OWL RL rule set are formulated in Datalog style, therefore they 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 a 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 modern architectures.

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

Example 2 For example, suppose that our initial query is

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

and that we have a generic database \( D \) and the OWL RL rule set 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 (cax-sco and prp-spo1 in the OWL RL rule set). 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.

With this process, we create a tree that has the original query as root and the rules and subqueries that might contribute to derive some answers as the internal nodes. This tree is normally referred as proof tree because it represents all the derivation steps that are taken to derive answers of our initial query (the root) starting from some existing facts (the leaves). In Figure 1 we report an example of such a 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 termination even in presence of recursive rules by memorizing in a global data structure all the subqueries already evaluated and
avoiding making 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 several 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 are 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 the fundamental properties of termination and completeness are still valid. After this, in Section 3.2, we will describe how we can exploit the pre-computation of some queries to increase the performance of backward-chaining.

### 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 as the original QSQ algorithm does, we do it in parallel by applying the rules on separate threads and in an asynchronous manner. For example, if we look back at Figure 1, the execution of rules cax-sco and prp-spl0 is performed concurrently by different threads. This execution strategy 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 to only remember which queries where already executed on the single paths of the tree. 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 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 store at every iteration all the intermediate derivations.

We report the algorithm using pseudocode in Algorithm 1. In the pseudocode, we use the relation ⊑ to define whether one query is more specific than another. In this case, all the results of the more specific query are contained in the answer set of the most generic one.

More formally, we can define it as follows:

Let \( t := (t_1, \ldots, t_n) \) and \( t' := (t'_1, \ldots, t'_n) \) be tuples with \( t_i, t'_i \in \text{TERM} \), i.e., each component is either a variable or a constant. Then \( t \) is an instance of \( t' \), \( t \subseteq t' \), if there is a substitution \( \sigma : \text{TERM} \rightarrow \text{TERM} \) such that \( \sigma(c) = c \) for each constant \( c \) and \( (\sigma(t'_1), \ldots, \sigma(t'_n)) = (t_1, \ldots, t_n) \).

Additionally, if \( R(t) \) and \( R(t') \) are atoms we define \( R(t) \sqsubseteq R(t') \) iff \( t \sqsubseteq t' \).

**Example 3** \((x, \text{TYPE}, \text{SYM}) \sqsubseteq (x, \text{TYPE}, y)\) where \( x, y \) are variables and \( \text{TYPE} \) and \( \text{SYM} \) are the abbreviation of Table 1.

Also \((x, \text{TYPE}, y) \sqsubseteq (y, \text{TYPE}, x)\).

If \( R(t) \sqsubseteq R(t') \) and \( R(t') \sqsubseteq R(t) \) then \( R(t) \) equals \( R(t') \) up to variable renaming.

The procedure main is the main function used to invoke the backward-chaining procedure for a given atomic query \( Q \). It returns the derived answers for the input query. The procedure consists of a loop in which the recursive function 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 Tmp. In each loop pass the latest results in Tmp and New are checked against the accumulated answers of the previous runs in Mat and Database. If nothing new could be derived then the loop terminates.

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

The function infer is the core of the backward-chaining algorithm. Using the function 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
The recursive call in line 19 will be fired only if there is no \(Q'\) in the set \(PrevQueries\) (see condition in line 15) such that \(Q \subseteq Q'\) and \(Q' \subseteq Q\) (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.

3.1.1. 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 3–6 and (ii) the recursive call in line 19. The first loop will continue until neither \(New\) nor \(Tmp\) 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 19 will be fired only if there is no \(Q'\) in the set \(PrevQueries\) (see condition in line 15) such that \(Q \subseteq Q'\) and \(Q' \subseteq Q\) (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 \( \text{infer}, R(a_1, \ldots, a_n) \) is a fact in \( P(\mathfrak{I}) \), the least fix-point model for the Datalog program \( P \) and the database \( \mathfrak{I} \). 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 for all answers to the input query \( Q \) derived under a given rule set \( \text{RuleSet} \) and a database \( \text{Database} \). 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 \( \text{infer}(Q, \emptyset) \) 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 \( r \in \text{RuleSet} \) 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 \( Q_k \sqsubseteq Q_i \)).

In this case \( \text{infer}(Q, \emptyset) \) will eventually produce the query \( Q_n \) (cf. lines 14-24, of Algorithm \( 1 \)).

Let \( Q(b_1, \ldots, b_m) \) be the atom 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 \( b_{i,k_i} \) of \( r_i \) unifies with the head of \( r_{i+1} \) and \( R(a_1, \ldots, a_n) \) unifies with some body-atom \( b_{n,\ell} \) 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 \( b_{i+1} \) 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 \( b_{n,\ell} \) of \( r_n \) where \( Q_n = \theta_n(b_{n,\ell}) \) 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 \( Q_1 \) 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 \( Q_0, \ldots, Q_i, Q_{i+1}, \ldots, Q_n \) 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 repeat-loop pass from which onwards \( R(a_1, \ldots, a_k) \) is returned by every query \( Q_n \) 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 \( n \) repeat-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 \( \text{infer}(Q, \emptyset) \) produces an unblocked query \( Q_n \) 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 26) by \( \text{infer} \) for all repeat-loop passes.

Assume the proof tree is of height \( > 0 \). Then there is a rule \( r : R(\bar{t}) \leftarrow R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m) \) 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 \( R_i(\beta(\bar{t}_i)) \) unifies with \( Q_i' := R_i(\theta(\bar{t}_i)) \). Since \( Q_n \) is not blocked, every \( Q_i' \) is a subsequent query of \( Q_n \).

By the induction hypothesis, for all \( i \in \{1, \ldots, n\} \), every \( Q_i' \) occurring in the computation yields \( R_i(\beta(\bar{t}_i)) \) after at most \( n \) repeat-loop passes. Hence \( R(a_1, \ldots, a_n) \) is returned by this particular \( Q_n \) at the very latest in the \( n \)-th repeat-loop pass and eventually added to \( \text{Mat} \) (cf. line 4) so that every subsequent query that unifies with \( R(a_1, \ldots, a_n) \) will return this fact as look-up in line 13.
By proving Proposition 2, we have shown that Algorithm 1 is complete in a way that given a generic ruleset and database, the algorithm is able to derive all possible conclusions that can be derived. However, in our approach this algorithm is invoked with a ruleset that is different from the one that should be used in first place. Therefore, we still need to prove that our entire approach is complete in a sense that an execution of this algorithm with the modified rule set retrieves the same results than an execution that uses the original ruleset (provided that a prematerialization is performed beforehand). This issue will be discussed in Section 4.

In the following section, we will conclude our discussion of our backward-chaining algorithm by describing how a generic set of the precalculated predicates can be used in the implementation to speed up the performance of reasoning at query time.

3.2. Exploiting the precomputation in the implementation of backward-chaining.

In the previous section, we made no difference in the description of our backward-chaining algorithm between subqueries that are precomputed or not. However, the pre-materialization of a selection of queries allows us to substantially improve the implementation and performance of backward-chaining performance by exploiting the fact that these queries can be retrieved with a single lookup.

In our implementation, these queries are stored in main memory so that the joins required by the rules can be efficiently executed. Also, the availability of the pre-materialized queries in memory allows us to implement a very efficient information passing strategy to reduce the size of the proof tree by identifying beforehand whether a rule can contribute to derive facts for a given query.

In fact, the pre-materialization can be used to determine early failures: Emptiness for queries which are subsumed by the pre-materialized queries can be cheaply derived since a lookup suffices. Therefore, when scheduling the derivation of rule body atoms, we give priority to those body atoms that potentially match these pre-materialized queries so that if these “cheap” body atoms do not yield any facts, the rule will not apply, and we can avoid the computation of the more expensive body atoms of the rule for which further reasoning would have been required.

To better illustrate this concept, we proceed with an example. Suppose we have the proof tree described in Figure 1. In this case, the reasoner can potentially apply rule prp-symp (concerning symmetric properties in OWL) to derive some triples that are part of the second body atom of rule prp-spo1.

However, in this case, Rule prp-symp will fire only if some of the subjects (i.e. the first component) of the triples part of $T(x, SPO, TYPE)$ will also be the subject of $T(x, TYPE, SYM)$. If both patterns are precalculated, then we know beforehand all the possible 'x', and therefore we can immediately perform an intersection between the two sets to see whether this is actually the case. If there is an intersection, then the reasoner proceeds executing rule prp-symp, otherwise it can skip its execution since it will never fire.

It is very unlikely that the same property appears in all the terminological patterns, therefore an information passing strategy that is based on the precalculated triple patterns is very effective in significantly reducing the tree size and hence improve the performance.

In the following section, we will focus on the pre-materialization phase, which purpose is to calculate the results for these subqueries in order to maintain our approach complete.

4. Hybrid Reasoning: Pre-Materialization

Before the user can query the knowledge base, our approach relies on a pre-materialization phase where we calculate some subqueries so that during query-time our backward-chaining algorithm is able to infer the entire derivation. We first formalize and discuss the pre-materialization algorithm and then we will 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.

4.1. Pre-Materialization

Let $\mathcal{I}$ 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 named $S_Q$ to $\mathcal{I}$ for each $Q \in L$ and populate these by setting $S_Q := \text{main}(Q)$ (for main cf. Algorithm 1) is that the efficiency of Algorithm 1 hinges upon that as many body atoms as possible are not 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 atom $S_Q(t_i)$ if $R_i(t_i) \subseteq Q$ (cf. page 5), 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 $S^3_Q$. Since this is repeated between each derivation until no new answers for any $Q \in L$ are yielded, this is equivalent to adding $S_Q(t_i) \leftarrow R_i(t_i)$ for each $R_i(t_i)$ with $Q = R_i(t_i)$ to $P'$ directly. This makes the procedure complete in the sense that, after termination of this algorithm, $S_Q$ contains all answers for the query $Q$ in the full materialization of $Database$ under $Ruleset$.

**Example 5** Take the altered program from Example 1 and add the appropriate $S_Q(Q) \leftarrow T(Q)$ to it. In this case we obtain

\[
\begin{align*}
T(a, p_1, b) & \leftarrow S(p, SPO, p_1) \land T(a, p, b) \\
T(x, SPO, y) & \leftarrow S(x, SPO, w) \land S(w, SPO, y) \\
S(x, SPO, y) & \leftarrow T(x, SPO, y)
\end{align*}
\]

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

\[
\begin{align*}
T(a, p_1, b) & \leftarrow T(p, SPO, p_1) \land T(a, p, b) \\
T(x, SPO, y) & \leftarrow T(x, SPO, w) \land T(w, SPO, y)
\end{align*}
\]

Algorithm 2 terminates and is sound in the sense that after Algorithm 2 has terminated $S_{R(i)}(a_1, \ldots, a_n)$ is in $\mathcal{J}$ whenever $R(a_1, \ldots, a_n)$ is an answer to $R(t_i)$ in the least fix-point model $P(\mathcal{J})$ of $P$ over $\mathcal{J}$.

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

**Proof** Assume for the sake of contradiction Algorithm 2 were not complete: Assume no new element could be derived in line 22 from the current state of the database $\mathcal{J}_0$ using the program $P'$ (defined as $NewRuleset$ in the pseudocode) but for some $Q \in L$, $main(Q)$ 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 undervied answer for any $Q \in L$ that is derived under the original program $P$.

Line 18 guarantees that all $S_Q^0 = Q_0$ and so program $P'$ is at this stage indistinguishable from $P$. Hence $main(Q)$ must derive the fact $R(a_1, \ldots, a_n)$ under $P'$, as well. A contradiction! Since $S_Q^0$ never shrinks during the pre-materialization process, Algorithm 2 is complete.

**Algorithm 2** Overall algorithm of the precomputation procedure: $L$ is a constant containing all queries that were selected for pre-materialization, $RuleSet$ is a constant containing a program $P$ and $Database$ represents $\mathcal{J}$.

```plaintext
for every $Q \in L$
    introduce a new predicate symbol $S_Q$ to $Database$
end for
for every rule $p : R_0(t_0) \leftarrow R_1(t_1) \land \ldots \land R_n(t_n)$ in $RuleSet$
    for every $Q \in L$
        if $R_i(t_i) \subseteq Q$ then
            replace $R_i(t_i)$ in $p$ with $S_Q(t_i)$
            end if
            end for
        add this (altered) rule to $NewRuleSet$
    end for
end for
Derivation := $\emptyset$
repeat
    $Database := Database \cup Derivation$
    for every $R(t) \in L$
        Perform $S_R(t) \leftarrow R(t)$ on $Database$
        end for
    for every $Q \in L$ (Derivation := Derivation $\cup main(Q)$ using $NewRuleSet$ as program on $Database$
        end for
    until Derivation $\subseteq Database$
```

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 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

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

base, 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.

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 the program \( P \). Let \( R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \land \ldots \land R_n(\bar{t}_n) \) be a rule in \( P \). The \( \bar{t}_i = (t_{i,1}, \ldots, t_{i,m_i}) \) 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 \( R_i(\bar{t}_i) \) is replaced by \( S(\bar{z}) \): Let \( \bar{z} := t_0 \cdot t_1 \cdot t_{i-1} \cdot t_{i+1} \cdot t_n \), i.e. the concatenation of all tuples except \( \bar{t}_i \) and let \( \bar{t} \) be some arbitrary tuple.\n
\[
q_0(\bar{z}) \leftarrow R_1(\bar{t}_1) \land \ldots \land R_i(\bar{t}_i) \land \ldots \land R_n(\bar{t}_n)
\]

\[
q_1(\bar{z}) \leftarrow R_1(\bar{t}_1) \land \ldots \land S(\bar{t}) \land \ldots \land R_n(\bar{t}_n)
\]

Now, the rule and its replacement fire under almost the same variable assignment iff \( q_0(\bar{z}) = q_1(\bar{z}) \), 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 \( \bar{t}_i \) and \( \bar{t} \). In this way we do not require, e.g., \( S^3 = R^P(\mathcal{J}) \). 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 \( T \) under \( P \) which would render our approach obsolete.

Also notice, that it is not sufficient to merely require \( q_0(\bar{t}_0) = q_1(\bar{t}_0) \), i.e. that both queries yield the same answer tuples \( \bar{t}_0 \) 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 \( R_0 := \{(a, b), (b, c), (b, b), (c, c)\} \). In the least fix-point model \( P(\mathcal{J}) \) of \( P \) we expect \( R_1^P(\mathcal{J}) = \{(a, b), (b, c), (a, c), (b, b), (c, c)\} \).

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 \( P(\mathcal{J}) \), i.e.

\[
q_0(x, z) \leftarrow R_1(x, y) \land R_0(y, z)
\]

\[
q_1(x, z) \leftarrow R_1(x, y) \land S(y, z)
\]

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 \( R_1^P(\mathcal{J}) = \{(a, b), (b, c), (b, b), (c, c)\} \).

We shall now show that substituting a body atom \( R_i(\bar{t}_i) \) by \( S(\bar{z}) \) under the condition that the queries in \((*)\) 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

\[
R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \land \ldots \land R_i(\bar{t}_i) \land \ldots \land R_n(\bar{t}_n)
\]

has, for some tuple \( \bar{t} \) in edb \( S \), been replaced by

\[
R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \land \ldots \land S(\bar{t}) \land \ldots \land R_n(\bar{t}_n).
\]

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

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

**Proof 4** In order to show the implication we assume \( q_0(\bar{z}) = q_1(\bar{z}) \). Let \( T_P \) and \( T_{P'} \) be the immediate consequence operators (mentioned on page
4) for each program. We show for all $k < \omega$ that if $Q(a_1, \ldots, a_m) \in T_P(\mathcal{I})$ then there is an $\ell < \omega$ such that $Q(a_1, \ldots, a_m) \in T_P^{\ell}(\mathcal{I})$ and vice versa. Since we start out from the same database $\mathcal{I}$ we have $T_P^{\ell}(\mathcal{I}) = T_P^{\epsilon}(\mathcal{I})$ which settles the base case.

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

If $R(\overline{t}_0) \leftarrow R_1(\overline{t}_1) \land \ldots \land R_n(\overline{t}_n) \in P'$, i.e. none of its body atoms where substituted, the induction hypothesis shows for each $j \in \{1, \ldots, n\}$ that we can find $\ell_j < \omega$ such that $R_j(\beta(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$. Let $\ell_0 := \max\{1\} \cup \{\ell_j \mid 1 \leq j \leq n\}$. Notice, that we add $\{1\}$ for the case where the rule body was empty.

In any case, we have $R_j(\beta(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$ for all $j \in \{1, \ldots, n\}$. Since all premises of this rule are satisfied, there is some $\ell < \omega$ such that $R_0(\beta(\overline{t}_0)) \in T_P^{\ell}(\mathcal{I})$.

If $R(\overline{t}_0) \leftarrow R_1(\overline{t}_1) \land \ldots \land R_n(\overline{t}_n) \not\in P'$ it is the rule where $R_0(\beta(\overline{t}_0))$ has been substituted with $S(\overline{t})$. For the assignment $\beta$ we now know $\beta(\overline{t}_0) = (a_1 \cdot \overline{t}_1 \cdots \overline{t}_i \cdots \overline{t}_{i+1} \cdots \overline{t}_n) \in q_0(\mathcal{I})P(\mathcal{I})$. Since $q_0(\mathcal{I})P(\mathcal{I}) = q_1(\mathcal{I})P(\mathcal{I})$ we know that there is some assignment $\beta'$, which coincides with $\beta$ on $\overline{t}_0 \cdot \overline{t}_1 \cdots \overline{t}_{i-1} \cdot \overline{t}_{i+1} \cdots \overline{t}_n$ such that $\beta'(\overline{t}_i) \in S^{P(\mathcal{I})}$.

Hence $R_j(\beta'(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$ for all $j \in \{1, \ldots, n\} \setminus \{i\}$ and $S(\beta'(\overline{t}_i)) \in T_P^{\ell}(\mathcal{I})$ since $S$ is an edb predicate. The induction hypothesis yields some $\ell_j < \omega$ for each $j \in \{1, \ldots, n\} \setminus \{i\}$ such that $R_j(\beta'(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$. Let $\ell_0 := \max\{0\} \cup \{\ell_j \mid 1 \leq j \leq n \text{ and } j \neq i\}$, then $R_j(\beta'(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$ for all $j \in \{1, \ldots, n\} \setminus \{i\}$ and $S(\beta'(\overline{t}_i)) \in T_P^{\ell}(\mathcal{I})$. Since all premises of this rule are satisfied, there is some $\ell < \omega$ such that $R_0(\beta(\overline{t}_0)) \in T_P^{\ell}(\mathcal{I})$. As $\beta$ coincides with $\beta'$ also on $\overline{t}_0$, i.e. $\beta'(\overline{t}_0) = (a_0, \ldots, a_m)$, we have in particular $R(a_0, \ldots, a_m) \in T_P^{\ell}(\mathcal{I})$.

This shows that for all predicates $Q$ we have $Q^{P(\mathcal{I})} \subseteq Q^{P(\mathcal{I})}$. For the converse we merely show the case of the substituted rule: Assume $R(a_1, \ldots, a_m) \in T_P^{\ell+1}(\mathcal{I})$ and there is an assignment $\beta'$ such that $\beta'(a_1, \ldots, a_m) = (a_1, \ldots, a_m) \land R_j(\beta'(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$ for all $j \in \{1, \ldots, n\} \setminus \{i\}$ as well as $\beta'(\overline{t}_i) \in S^{P(\mathcal{I})}$.

The induction hypothesis yields for each $j \in \{1, \ldots, n\} \setminus \{i\}$ some $\ell_j < \omega$ with $R_j(\beta'(\overline{t}_j)) \in T_P^{\ell}(\mathcal{I})$. Since $S$ is an edb predicate for $P$, we have $S(\beta'(\overline{t}_i)) \in T_P^{\ell}(\mathcal{I})$. Hence for $\ell_0 := \max\{0\} \cup \{\ell_j \mid 1 \leq j \leq n \text{ and } j \neq i\}$ we have $R_j(\beta'(\overline{t}_j)) \in T_P^{\ell_0}(\mathcal{I})$ for all $j \in \{1, \ldots, n\} \setminus \{i\}$ and $S(\beta'(\overline{t}_i)) \in T_P^{\ell_0}(\mathcal{I})$.

This implies $\beta'(\overline{t}_0 \cdot \overline{t}_i \cdot \overline{t}_{i+1} \cdots \overline{t}_n) \in q_1(\mathcal{I})P(\mathcal{I})$ and since $q_0(\mathcal{I})P(\mathcal{I}) = q_1(\mathcal{I})P(\mathcal{I})$ there is an assignment $\beta'$ coinciding on $(\overline{t}_0 \cdot \overline{t}_i \cdot \overline{t}_{i+1} \cdots \overline{t}_n)$ with $\beta'$ such that $R_0(\beta'(\overline{t}_0)) \in T_P^{\ell_0}(\mathcal{I})$ for some $\ell_0 < \omega$.

Let $\ell_1 := \max\{\ell_0, j_0\}$ then $R_j(\beta'(\overline{t}_j)) \in T_P^{\ell_1}(\mathcal{I})$ for all $j \in \{1, \ldots, n\}$. Since all premises of the rule $R_0(\overline{t}_0) \leftarrow R_1(\overline{t}_1) \land \ldots \land R_n(\overline{t}_n)$ are satisfied, there is some $\ell < \omega$ such that $R_0(\beta(\overline{t}_0)) \in T_P^{\ell}(\mathcal{I})$, which shows, as $\beta$ coincides on $\overline{t}_0$ with $\beta'$ that $R(a_0, \ldots, a_m) \in T_P^{\ell}(\mathcal{I})$.

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

It now becomes clear, how Algorithm 2 and Proposition 4 fit together: For a given database $\mathcal{I}$ and a list of atomic queries $L$, Algorithm 2 computes for each $Q \in L$ the query answers under the program $P$, which are stored in the relation $S_Q$. These $S_Q$ are edbs for $P$.

Let now $r : R_0(\overline{t}_0) \leftarrow R_1(\overline{t}_1) \land \ldots \land R_n(\overline{t}_n)$ be a rule in this program and $Q \in L$ an atomic query s.t. $R_i(\overline{t}_i) \subseteq Q$, then $Q = R_i(\overline{t}_i)$ such that $\overline{t}_i \subseteq \overline{t}$ by definition of $\subseteq$. Correctness of Algorithm 2 yields $R(\overline{t}_i)^{P(\mathcal{I})} = S_Q(\overline{t}_i)^{P(\mathcal{I})}$ and hence that $q_0(\mathcal{I})P(\mathcal{I}) = q_1(\mathcal{I})P(\mathcal{I})$.

Proposition 4 guarantees that the substitution of $R_i(\overline{t}_i)$ by $S_Q(\overline{t}_i)$ in rule $r$ is harmless w.r.t. $\mathcal{I}$. By applying this argument iteratively, one eventually obtains a program $P'$ in which all pre-computed atoms have been replaced and which yields the same materialization for $\mathcal{I}$ as $P$.

In the following section we shall apply this rewriting to the OWL RL rule set.

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 completeness of our approach.

We will now discuss the implementation of the OWL RL rules using our approach. The official OWL RL rule set contains 78 rules, for which the reader is

**Initial assumptions.** First of all, we exclude some rules from our discussion and implementation for various reasons. These are:

- All the rules whose purpose is to derive an inconsistency, i.e. rules with predicate *false* in the head of the rule. We do not consider them because our purpose is to derive new triples rather than identify an inconsistency;
- All the rules which have an empty body. These rules cannot be triggered during the unfolding process of backward-chaining. These rules include the ones who encode the semantics of datatypes, therefore our implementation does not support datatypes;
- The rules that exploit the *owl:sameAs* transitivity and symmetry\(^4\). These rules require a computation that is too expensive to perform at query time since they can be virtually applied to every single term of the triples. These rules can be implemented by computing the *sameAs* closure and maintaining a consolidation table.\(^5\)

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 2 we report the list of the patterns that are pre-materialized using our method described in Section 4.

Singling out exactly those triple patterns from Table 2 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.

These characteristics make the set of inferred schema triples the ideal candidate to be pre-materialized. 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

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

where all answers to \(T(?c_1 \ SCO ?c_2)\) are contained in \(S_{sco}\). Then \(T(x, SCO, y)\) can be replaced by \(\top\) 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.

**On the implementation of RDF lists.** Some of the inference rules in the OWL RL rule set use RDF lists to define a variable number of antecedents. The RDF lists cannot be represented in Datalog in a straightforward way since they rely on *rdf: first* and *rdf: next* triples to

\(^4\)The list of these rules is reported in Table 4 of [11].

\(^5\)This procedure is explained in detail in [16].

| OWL RL | RDFS | P.D.
|---|---|---
| (≥X rdfs:subPropertyOf ≥Y) | (≥Y rdfs:subClassOf ≥X) | (≥X rdfs:domain ≥Y) |
| (≥X rdfs:range ≥Y) | (≥P rdf:type owl:FunctionalProperty) | (≥Y owl:inverseOf ≥X) |
| (≥Y owl:allValuesFrom ≥X) | (≥P rdf:type owl:InverseFunctionalProperty) | (≥Y owl:inverseOf ≥Y) |
| (≥Y owl:domain ≥X) | (≥P rdf:type owl:TransitiveProperty) | (≥Y owl:domain ≥X) |
| (≥P rdf:type owl:SymmetricProperty) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥Y owl:range ≥X) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥Y owl:range ≥X) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥Y owl:range ≥X) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥P rdf:type owl:FunctionalProperty) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥Y owl:range ≥X) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥Y owl:range ≥X) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |
| (≥Y owl:range ≥X) | (≥Y owl:range ≥Y) | (≥Y owl:range ≥X) |

Table 2: Triple patterns that are precalculated considering the OWL RL rules.
We will now analyze each of these three cases below.

In our implementation, at every step of the pre-materialization procedure, we launch two additional queries to retrieve all the (inferred and explicit) \texttt{rdf:first} and \texttt{rdf:rest} triples with the purpose of construct such lists. Once we have collected them, we perform a join with the other schema triples, and determine the sequence of elements by repetitively joining the \texttt{rdf:first} and \texttt{rdf:rest} triples. After this operation is completed, from the point of view of the Datalog program RDF lists appear as simple list of elements and are used according to the various rule logics. For example, if the rule requires a matching of all the elements of the list with the other antecedents, then the rule executor will first use the first element to perform the join, then it will use the second, and so on, until all the elements are considered.

5.1. Detecting duplicate derivation in OWL RL

Since the OWL RL fragment consists of a large number of rules, there is a high possibility that the proof tree contains branches that lead to the same derivation. Detecting and 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 rule set. 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 tables in the following list to the OWL RL rule set in [11]:

- \texttt{prp-eqpl} and \texttt{prp-eqp2} from Table 5
- \texttt{cax-eqc1} and \texttt{cax-eqc2} from Table 7
- \texttt{prp-inv1} and \texttt{prp-inv2} from Table 5.

We will now analyze each of these three cases below.

Let \( S^3_{eqp} \) be the pre-materialization of the triple pattern \( T(\exists x, EQP, ?y) \). Rules \texttt{scm-eqpl} and \texttt{scm-eqp2} render \( S^3_{eqp} \) symmetric. Hence

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

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 \texttt{scm-eqp2} can be replaced by \texttt{scm-eqpl}, effectively deleting \texttt{scm-eqp2} from the rule set.

Similarly, rules \texttt{scm-eqc1} and \texttt{scm-eqc2} render the results of the pre-materialized query \( T(\exists x, EQC, ?y) \), symmetric.

In contrast, the pre-materialized query \( T(\exists x, INV, ?y) \) is not symmetric. However, we can first observe that Proposition 4 allows us 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 defines the idb atom \( T(q, INV, p) \) into the harmless edb \( S_{inv} \), for which \( S^3_{inv} \) contains all answers to the query \( T(\exists x, INV, ?y) \). Let this new program be called \( P'' \). Further, let \( P''' \) be the program where both rules have been replaced by

\[
T(x, p, y) \leftarrow T(x, q, y) \land S''_{inv}(p, INV, q)
\]

with \( S''_{inv} \) being the symmetric closure of \( S^3_{inv} \). 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 \texttt{prp-inv1} and \texttt{prp-inv2} by one rule under the condition that we pre-materialize the symmetric closure of \( T(\exists x, INV, ?y) \).

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 \texttt{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 \texttt{scm-sco} and \texttt{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 can apply a special algorithm when the system is starting up and it is initializing the ruleset. A complete description of this algorithm is outside the scope of this chapter and we will simply illustrate the main idea behind it.
Basically, this algorithm compares each rule with all the others in order to identify under which conditions the two will produce the same output to a given query. For example, the rules \( \text{scm-sco} \) and \( \text{prp-trp} \) will produce the same derivation if (i) the input contains the triple \( T(\text{SCO}, \text{TYPE}, \text{TRANS}) \) and if (ii) there is a query with \( \text{SCO} \) as a predicate.

In order to verify this is the case, the algorithm checks whether the triple \( T(\text{SCO}, \text{TYPE}, \text{TRANS}) \) exists in the input and there is a matching on the position of the variables in the two rules (if one rule contains more variables than the other, then the algorithm will substitute the corresponding terms). If such matching exists, then the two rules are equivalent. In our example, the algorithm will find out that the rule \( \text{prp-trp} \) is equivalent to \( \text{scm-sco} \) if we replace \( ?p \) with \( \text{SCO} \). Therefore, if there is an input query with \( \text{SCO} \) as predicate, the system will execute only one of the two rules, avoiding in this way a duplicated derivation.

### 6. Evaluation

We have implemented our approach in a Java prototype that we called QueryPIE\(^6\) and we evaluated the performance using one machine of the DAS-4 cluster\(^7\), that 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 configured in RAID-0 mode.

We used two datasets as input. LUBM [7], which is one of the most popular benchmarks for OWL reasoning and LLD (Linked Life Data)\(^8\), which is a curated collection of real-world datasets in the bioinformatics domain.

LUBM allows us 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. Finally, in Section 6.3, we present a more general discussion on the results that we obtained.

#### 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 3. 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 three 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 intend to compare the cost of performing the partial closure against 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 that is able to scale to the input size that we consider.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Reasoning time</th>
<th>N. iterations</th>
<th>N. derived triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM</td>
<td>1.0s</td>
<td>4d4h16m</td>
<td>4</td>
</tr>
<tr>
<td>LLD</td>
<td>16m</td>
<td>5d10h45m</td>
<td>7</td>
</tr>
</tbody>
</table>

#### Table 3

Execution time of the pre-materialization algorithm compared to a full closure.

---

\(^6\)We intend to release all the software with an open-source license and plan to reference it in case this paper will be accepted.

\(^7\)http://www.cs.vu.nl/das4

\(^8\)http://www.linkedlifedata.com/
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 3. 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 advantage of our approach in terms of pre-materialization cost. However, such advantage comes at a price: while after a complete materialization reasoning is no longer needed, in our case we still have to perform some inference at query time. The impact of this operation on the query-time performance is analyzed in the next section.

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 4.

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 5 the execution time obtained launching the selected example queries in Table 4. In the second and third columns we report both the cold and warm runtime. With cold runtime we identify the

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

Table 4
List of example queries

---

9 Notice that the reported runtime does not include the time required to compress/decompress the numerical terms to their string counterpart.
Table 5
Runtime of the queries in Table 4 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>60.43</td>
<td>6.39</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1099.28</td>
<td>129.31</td>
<td>463</td>
</tr>
<tr>
<td>3</td>
<td>49.18</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>73.06</td>
<td>11.17</td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td>118.71</td>
<td>13.97</td>
<td>1480</td>
</tr>
<tr>
<td>6</td>
<td>4026.27</td>
<td>2509.27</td>
<td>1599987</td>
</tr>
<tr>
<td>7</td>
<td>228.26</td>
<td>214.57</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>23.74</td>
<td>6.29</td>
<td>4466</td>
</tr>
<tr>
<td>9</td>
<td>7064.04</td>
<td>609.4</td>
<td>140</td>
</tr>
<tr>
<td>10</td>
<td>2535.38</td>
<td>1103.48</td>
<td>28446</td>
</tr>
<tr>
<td>11</td>
<td>2613.37</td>
<td>1883.14</td>
<td>8546</td>
</tr>
<tr>
<td>12</td>
<td>2334.70</td>
<td>2059.20</td>
<td>1187944</td>
</tr>
</tbody>
</table>

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 thirty 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 11 the program had access to the data indices about 15000 times.

From the results reported in Table 5 we can make few considerations. First, we notice that the cold runtime is in general significantly higher 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 337 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) There is a large difference in the runtime observed in our experiments. In the worst case the absolute runtime is in the range of few seconds, while in the best cases the performance is in the order of dozens of milliseconds. However, even in the worst case the system allows an interactive usage since few seconds are acceptable in most scenarios.

In Section 6.2.3 we will compare such response times with the ones without reasoning in order to have a better overview of the overall performance and understand what is the overhead induced by reasoning at query time.

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 such as 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
Table 7

<table>
<thead>
<tr>
<th>Query</th>
<th>Only Lookup</th>
<th>RDFS</th>
<th>pD*</th>
<th>OWL RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.81</td>
<td>0.83</td>
<td>1.88</td>
<td>1.79</td>
</tr>
<tr>
<td>2</td>
<td>0.82</td>
<td>1.51</td>
<td>1.56</td>
<td>2.83</td>
</tr>
<tr>
<td>3</td>
<td>0.82</td>
<td>0.83</td>
<td>3.55</td>
<td>2.72</td>
</tr>
<tr>
<td>4</td>
<td>0.88</td>
<td>0.94</td>
<td>2.01</td>
<td>2.32</td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
<td>1.61</td>
<td>7.01</td>
<td>4.95</td>
</tr>
<tr>
<td>6</td>
<td>405.42</td>
<td>418.38</td>
<td>2605.68</td>
<td>2630.08</td>
</tr>
<tr>
<td>7</td>
<td>0.77</td>
<td>0.79</td>
<td>176.19</td>
<td>1.26</td>
</tr>
<tr>
<td>8</td>
<td>1.96</td>
<td>1.89</td>
<td>6.23</td>
<td>6.34</td>
</tr>
<tr>
<td>9</td>
<td>0.84</td>
<td>0.90</td>
<td>262.7</td>
<td>46.53</td>
</tr>
<tr>
<td>10</td>
<td>7.90</td>
<td>7.29</td>
<td>212.57</td>
<td>115.16</td>
</tr>
<tr>
<td>11</td>
<td>1.85</td>
<td>1.93</td>
<td>200.55</td>
<td>8.35</td>
</tr>
<tr>
<td>12</td>
<td>338.14</td>
<td>337.41</td>
<td>2129.49</td>
<td>2044.34</td>
</tr>
</tbody>
</table>

Table 7

Runtime (in ms.) of the example queries changing the rule set.

Table 6

<table>
<thead>
<tr>
<th>Query</th>
<th># Leaves proof tree</th>
<th>Reduction ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without precomp.</td>
<td>Our approach</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 6

Estimation of the reduction of the proof tree caused by the pre-materialization algorithm.

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 further 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 remaining 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 warm runtime in Table 7.

We proceeded as follows: we first launch the queries, deactivating all rules at query time, and state their execution 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

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 deactivate all the optimizations, and therefore in reality 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 precisely captures small amounts of inferences that contribute substantially to the reasoning costs because they are being used very often.
column), then the $pD*$ rules and finally the OWL RL ones.

The results reported under the “Ins.” columns were calculated differently. In fact, in the previous experiments the number of retrieved results for a specific query might differ because we changed the rule set and this can influence the general performance. To maintain the number of results constant, we have repeated the same experiment adding to the knowledge base all the possible results so that even if reasoning is not activated the same number of results is retrieved (“Ins.” means insertion).

From the results presented in the table, we notice that in both cases (“Ins” and “No Ins.”) the response time progressively increases as we include 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 storing the results into main memory during the query execution to ensure the completeness of the backward-chaining algorithm. This operation is clearly a non-negligible contributor to the overall performance.

We can compare the response times reported in the third column with the ones of the penultimate column 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 three order of magnitude slower. In case the query needs to process and/or return many triples, the difference is certainly significant. However, the response time is still in the order of the hundreds of milliseconds, from the user perspective, the difference is less noticeable and more easily tolerated especially considering that a large precomputation phase is no longer needed.

6.3. Discussion

In this paper, we chose to evaluate our method using the most common and large-scale datasets currently available in order to evaluate how hybrid reasoning would perform on current data and realistic queries.

The measurements that we report have shown that large scale OWL RL reasoning is indeed possible, even on a relatively modest computer architecture. However, we must point out that these datasets do not (yet) use all the features introduced with the OWL 2 language and, to the best of our knowledge, there is no large-scale benchmark that extensively uses these new features.

Therefore, there is a remaining open question on what the performance would be on an input that exploits all the features of the OWL 2 language. While such problem is beyond the scope of this article, we can make some considerations by looking at the experiments here presented.

First of all, our approach most likely would not able to guarantee the same response time in the worst-case scenario. This is not particularly due to a limitation of our method, but rather to the high computational complexity intrinsically required by reasoning.

However, even without looking at the complete worst-case scenario (which is very unlikely to happen in practice), there can be other cases where the performance could be significantly worse. In our experiments, we noticed that as the size of the proof tree increases, so does the potential derivation of duplicates due to the potential higher number of combinations. In Section 5.1, we tackled this problem by proposing some initial algorithms to limit the number of duplicates. However, our work in this respect is still initial and further research on this particular aspect might become necessary in order to scale not only in terms of input size but also in terms of reasoning complexity.

Summarizing, we observe in our evaluation that fairly complex reasoning can be performed rather quickly (in a matter of few seconds in the worst case) on realistic queries and on large data. However, the reader should keep in mind that there could be worst-case scenarios (which do not seem to appear on current data) where the performance is significantly worse, and this is mainly due to the theoretical high worst-case complexity that is inherently present in the reasoning process.

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 pro-
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 RDFS 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 rule set 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 use case 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 most of the OWL RL rules. The performance analysis shows that the query response-time for our approach is 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 such 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 complex backward-chaining reasoning over realistic OWL 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 and debugging the prototype used in the evaluation.

A sincere thanks goes to the reviewers of this article, Aidan Hogan, Matthias Knorr, and Raghava Mutharaju, who have significantly improved the quality of this article with a careful review of the work. This work was partly supported by the LarKC project (EU FP7-215535) and by the COMMIT project as well as the SEALS Project.

References