

VU Research Portal

Hybrid reasoning on OWL RL

Urbani, J.; Piro, R.; van Harmelen, F.A.H.; Bal, H.E.

published in

Semantic Web
2013

DOI (link to publisher)

[10.3233/SW-130120](https://doi.org/10.3233/SW-130120)

document version

Early version, also known as pre-print

[Link to publication in VU Research Portal](#)

citation for published version (APA)

Urbani, J., Piro, R., van Harmelen, F. A. H., & Bal, H. E. (2013). Hybrid reasoning on OWL RL. *Semantic Web*. <https://doi.org/10.3233/SW-130120>

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:

vuresearchportal.ub@vu.nl

Hybrid reasoning on OWL RL

Editor(s): Pascal Hitzler, Kno.e.sis Center, Wright State University, USA

Solicited review(s): Aidan Hogan, Digital Enterprise Research Institute, National University of Ireland, Galway; Matthias Knorr, Universidade Nova de Lisboa, Portugal; Raghava Mutharaju, Kno.e.sis Center, Wright State University, USA; Peter F. Patel-Schneider, Nuance Communications, USA

Jacopo Urbani ^{a,*}, Robert Piro ^b, Frank van Harmelen ^a and Henri Bal ^a

^a *Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands*
E-mail: {jacopo,frankh,bal}@cs.vu.nl

^b *Department of Computer Science, University of Oxford, United Kingdom*
E-mail: 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 pre-computation, 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 data sets.

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 (most of) 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 data sets 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 pre-computation takes only a few minutes against the hours (or even days) necessary for a full materialization and that (ii) the remaining overhead introduced by reasoning still allows atomic 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 the literature [11,20,27] 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 together with the original input.

*Corresponding author. E-mail: jacopo@cs.vu.nl.

1 While these methods exhibit good scalability because
 2 they can efficiently exploit computational parallelism,
 3 they have several disadvantages which compromise
 4 their use in real-world scenarios. First, they cannot effi-
 5 ciently deal with small incremental updates since they
 6 have to compute the complete materialization anew.
 7 Second, they become inefficient if the user is only in-
 8 terested in a small portion of the entire input because
 9 forward-chaining needs to calculate all derivations.

10 Unlike forward-chaining, backward-chaining ap-
 11 plies only inference rules depending on a given query.
 12 In this case, the computations required to determine
 13 the rules that need to be executed often become too ex-
 14 pensive for interactive applications. Thus, backward-
 15 chaining has until now been limited to either small data
 16 sets (usually in the context of expressive DL reasoners)
 17 or weak logics (RDFS inference).

18 In this paper, we propose a method which material-
 19 izes a fixed set of selected queries, *before* query time,
 20 while applying backward chaining *during* query time.
 21 This *hybrid* approach is a trade-off between a reduc-
 22 tion in rule applications at query time and a small,
 23 query independent computation of data before query
 24 time. Our backward-chaining algorithm exploits the
 25 parallel computing power of modern architectures and
 26 uses at query time a partial materialization of some se-
 27 lected queries to reduce the computation.

28 We will show that our backward chaining algorithm
 29 is correct, i.e. it terminates, is sound and complete. We
 30 shall argue that the correctness is not dependent on a
 31 particular rule set but holds for any Datalog program.

32 For the implementation and evaluation, however, we
 33 apply our method considering the semantics of the
 34 OWL RL fragment, which is one of the most recently
 35 standardized OWL profiles designed to work on a large
 36 scale.

37 To this end, we have implemented the backward-
 38 chaining algorithm using the results of the pre-materia-
 39 lized queries in an experimental prototype called
 40 QueryPIE and tested the performance using artificial
 41 and realistic data sets of a size between five and ten
 42 billion triples. The evaluation shows that we are able to
 43 perform OWL reasoning using one machine equipped
 44 with commodity hardware which keeps the response
 45 time often below one second.

46 This paper is a revised and improved version of our
 47 initial work that was presented in [19]. More specifi-
 48 cally, it extends the initial version that targets the pD^*
 49 fragment to one that supports most of the OWL RL
 50 rules, which are officially standardized by W3C. Also,
 51 this paper provides a theoretical analysis of the ap-

52 proach proving its correctness w.r.t. the considered rule
 53 set and presents an improved explanation and evalua-
 54 tion over larger data sets.

55 The remainder of this paper is organized as follows:
 56 Section 2 presents notations and notions used through-
 57 out the paper. In Section 3 we introduce the reader to
 58 our problem and provide a high level overview of our
 59 approach.

60 Next, in Section 4, we describe the backward-
 61 chaining algorithm that is used in our method to cal-
 62 culate the inference. Section 5 formalizes the pre-
 63 computation algorithm of hybrid reasoning and proves
 64 its correctness. Section 6 focuses on the execution of
 65 most of the OWL 2 RL/RDF rule set (henceforth sim-
 66 ply referred to as OWL RL rule set), specifying which
 67 rules are excluded and presenting a series of optimiza-
 68 tions to improve the performance on a large input.¹

69 In Section 7 we present an evaluation of our ap-
 70 proach using atomic queries on both realistic and arti-
 71 ficial data. In Section 8 we report on related work.
 72 Finally, Section 9 concludes and gives directions for
 73 future work.

74 2. Preliminaries

75 In this section, we set out notational conventions and
 76 briefly recall some well-known notions from Database
 77 theory, where we mainly follow [1, Chapter 12].

78 The algorithms we present run on Datalog pro-
 79 grams, since the OWL RL inference rules are formu-
 80 lated in Datalog style; therefore most rules of our se-
 81 lected rule set, with a few exceptions (cf. Section 6,
 82 On the implementation of RDF lists), can be trivially
 83 rendered into a Datalog program.

84 Throughout this paper, we use abbreviations to in-
 85 dicate well-known URIs for reasons of space.² In our
 86 notations, we use as a convention fixed-width charac-
 87 ters to denote constant terms (e.g. SPO) as well as ital-
 88 ics for Datalog variables and predicate names (e.g. a
 89 or T). Note that in SPARQL [16] and the official OWL
 90 RL documentation variables are indicated with a pre-
 91 ceding “?” (e.g. $?a$).

92 For two functions $f : A \rightarrow B$ and $g : B' \rightarrow C$
 93 with $B \subseteq B'$ we denote with $g \circ f$, pronounced g after
 94 f , the function $A \rightarrow C : x \mapsto g(f(x))$. For a set of

100 ¹Note that by excluding some rules our approach is incomplete
 101 w.r.t. the official OWL RL specification.

102 ²Table 1 reports a list of all the abbreviations used in this paper.

1 functions $G := \{g \mid g : B_g \rightarrow C_g\}$ with $B \subseteq B_g$,
 2 for all $g \in G$ we define $G \circ f := \{g \circ f \mid g \in G\}$.

3 In Datalog, a signature SIG is a finite set of symbols
 4 which is the disjoint union of the set CONS of constants
 5 and the set PRED of predicate symbols. Each
 6 predicate symbol is associated with its *arity*, a positive
 7 natural number. A *database* \mathcal{J} for SIG is a pair consisting
 8 of a finite set $dom \mathcal{J}$, the *domain*, and an interpretation
 9 function $\cdot^{\mathcal{J}}$ whose domain is SIG.

10 If $R \in \text{PRED}$ is an n -ary predicate symbol, then
 11 $R^{\mathcal{J}}$, where $R^{\mathcal{J}} \subseteq (dom \mathcal{J})^n$, denotes the n -ary relation
 12 named R in the database \mathcal{J} . If $c \in \text{CONS}$ then $c^{\mathcal{J}} \in$
 13 $dom \mathcal{J}$. For our purposes, we assume $\text{CONS} = dom \mathcal{J}$
 14 and $c^{\mathcal{J}} = c$ for all $c \in \text{CONS}$.

15 With VAR we denote a countably infinite set of variables
 16 where $\text{VAR} \cap \text{SIG} = \emptyset$. Thus, the set of all terms
 17 is $\text{TERM} = \text{VAR} \cup \text{CONS}$. For every term tuple \bar{t} we
 18 denote with $Var(\bar{t})$ the set of all variables in \bar{t} .

19 If $R \in \text{PRED}$ is an n -ary predicate symbol and \bar{t}
 20 is an n -ary term tuple, then $R(\bar{t})$ is called *atom*. An
 21 atom $R(\bar{t})$ is called *ground atom* if $\bar{t} \subseteq \text{CONS}$. If \mathcal{J} is a
 22 database over SIG and $R \in \text{PRED}$ we write $R(\bar{a}) \in \mathcal{J}$
 23 if $R(\bar{a})$ is a ground atom and $\bar{a} \in R^{\mathcal{J}}$. We continue
 24 Var on atoms by setting $Var(R(\bar{t})) := Var(\bar{t})$ for every
 25 atom $R(\bar{t})$.

26 A *substitution* is a mapping $\theta : V \rightarrow \text{TERM}$
 27 where $V \subseteq \text{VAR}$. The domain V of θ is also denoted
 28 $dom \theta$. We call the substitution θ *assignment*,
 29 if $\theta(V) \subseteq \text{CONS}$ and θ is called *variable renaming*
 30 if $\theta(V) \subseteq \text{VAR}$ and θ is injective. $\theta_\varepsilon : \emptyset \rightarrow \emptyset$ is
 31 the *empty substitution*. Every substitution θ has a continuation
 32 $\tilde{\theta}$ on terms, where $\tilde{\theta}(t) = \theta(t)$ if $t \in V$
 33 and $\tilde{\theta}(t) = t$ if $t \in \text{TERM} \setminus V$. Henceforth, we
 34 will denote θ but always implicitly refer to its continuation
 35 $\tilde{\theta}$. Similarly we apply substitutions or rather
 36 their continuations to term tuples $\theta(t_1, \dots, t_n) :=$
 37 $(\theta(t_1), \dots, \theta(t_n))$ and atoms $\theta(R(\bar{t})) := R(\theta(\bar{t}))$.

38 We allow ourselves to represent a substitution θ as
 39 a set $\{t_0/t_1 \mid t_0 \in dom \theta \text{ and } \theta(t_0) = t_1\}$. The set-
 40 representation of θ_ε is simply \emptyset .

41 Using substitutions as results for database look-ups
 42 is not unusual and joins (\bowtie) over sets of substitutions
 43 are particularly used in SPARQL [16]. To define joins
 44 in this way, we need the following notions:
 45 A substitution θ_0 is *compatible* with a substitution θ_1
 46 if $\theta_0(t) = \theta_1(t)$ for all $t \in dom \theta_0 \cap dom \theta_1$. For
 47 each pair of compatible substitutions θ_0, θ_1 we set
 48 $\theta_0 \cup \theta_1$ to be the substitution which corresponds to
 49 the union of their set-representations. θ_ε is compatible
 50 with every substitution θ and is neutral in the sense
 51 that $\theta_\varepsilon \cup \theta = \theta \cup \theta_\varepsilon = \theta$. For sets Θ_0 and Θ_1 of sub-

stitutions we define $\Theta_0 \bowtie \Theta_1 := \{\theta_0 \cup \theta_1 \mid \theta_0 \in$
 $\Theta_0 \text{ compatible with } \theta_1 \in \Theta_1\}$.

Example 1. As an example of a join between substitutions,
 consider the rule (cax-sco)

$$T(x, \text{TYPE}, z) \leftarrow T(x, \text{TYPE}, y), T(y, \text{SCO}, z)$$

and let J, B, S, P stand for John, Brother, Sister and
 Person. We calculate the join of the two sets of substitutions
 where the first one can be considered to be the result of the
 query $T(x, \text{TYPE}, y)$ and the second one to be the result of
 $T(y, \text{SCO}, z)$ respectively:

$$\begin{aligned} & \{\{x/J, y/B\}\} \bowtie \{\{y/S, z/P\}, \{y/B, z/P\}\} \\ & = \{\{x/J, y/B, z/P\}\} \end{aligned}$$

If the resulting substitution is applied to $T(x, \text{TYPE}, z)$,
 we obtain from John being a Brother and Brothers and
 Sisters being Persons that John is of type Person.

Let $R(\bar{t}_0)$ and $R(\bar{t}_1)$ be two atoms with $Var(\bar{t}_0) \cap$
 $Var(\bar{t}_1) = \emptyset$. A *unifier* for $R(\bar{t}_0)$ and $R(\bar{t}_1)$ is a substitution
 $\theta : (Var(\bar{t}_0) \cup Var(\bar{t}_1)) \rightarrow (\bar{t}_0 \cup \bar{t}_1)$ such that
 $\theta(R(\bar{t}_0)) = \theta(R(\bar{t}_1))$. A *most general unifier* (MGU)
 of two atoms $R(\bar{t}_0)$ and $R(\bar{t}_1)$ is a unifier θ for $R(\bar{t}_0)$
 and $R(\bar{t}_1)$ such that for each unifier θ' of $R(\bar{t}_0)$ and
 $R(\bar{t}_1)$ there is a substitution σ with $\theta' = \sigma \circ \theta$.³ It is
 decidable whether or not a unifier for two given atoms
 exists. If it exists, an MGU exists and can be computed.

A Datalog *query* is an expression $q(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge$
 $\dots \wedge R_n(\bar{t}_n)$ where \bar{t}_0 is a term tuple, $R_i(\bar{t}_i)$ is
 an atom for each $i \in \{1, \dots, n\}$ and $Var(\bar{t}_0) \subseteq$
 $\bigcup_{1 \leq i \leq n} Var(\bar{t}_i)$. We omit \bar{t}_0 from $q(\bar{t}_0)$ if it is clear or
 not of interest. We call $q(\bar{a})$ an *answer* to $q(\bar{t}_0)$ w.r.t.
 \mathcal{J} if there is an assignment β with $\beta(\bar{t}_0) = \bar{a}$ and for
 all $i \in \{1, \dots, n\}$ we have $\beta(\bar{t}_i) \in R_i^{\mathcal{J}}$. The set of all
 answers to $q(\bar{t}_0)$ w.r.t. \mathcal{J} is denoted as $q(\bar{t}_0)^{\mathcal{J}}$.

We call a query *atomic* if $n = 1$ and will refer to it
 by its sole atom $R_1(\bar{t}_1)$. Answers in \mathcal{J} to an atom query
 $R_1(\bar{t}_1)$ are ground atoms $R_1(\bar{a}) \in \mathcal{J}$ such that \bar{t}_1 and
 \bar{a} unify.

Let \bar{t} and \bar{t}' be term tuples of the same length. Then
 \bar{t} is an *instance* of \bar{t}' , $\bar{t} \sqsubseteq \bar{t}'$, if there is a substitution σ
 such that $\sigma(\bar{t}') = \bar{t}$. Additionally, if $R(\bar{t})$ and $R(\bar{t}')$ are
 atoms we define $R(\bar{t}) \sqsubseteq R(\bar{t}')$ and say $R(\bar{t}')$ is *at least*
as general as $R(\bar{t})$ iff $\bar{t} \sqsubseteq \bar{t}'$. $R(\bar{t})$ equals $R(\bar{t}')$ up to
 variable renaming iff $R(\bar{t}) \sqsubseteq R(\bar{t}')$ and $R(\bar{t}') \sqsubseteq R(\bar{t})$.

Example 2. $(x, \text{TYPE}, \text{SYM}) \sqsubseteq (x, \text{TYPE}, y)$ where
 x, y are variables and TYPE and SYM are the abbrevi-
 ation of Table 1. Also $(x, \text{TYPE}, y) \sqsubseteq (y, \text{TYPE}, x)$.

³In literature, substitutions are used in post-fix notation, hence the
 condition for being an MGU is denoted there as $\theta' = \theta\sigma$.

Table 1

List of abbreviations for common URIs used in this paper

Abbreviation	Full text
TYPE	<i>rdf:type</i>
SCO	<i>rdfs:subClassOf</i>
SPO	<i>rdfs:subPropertyOf</i>
EQC	<i>owl:equivalentClass</i>
EQP	<i>owl:equivalentProperty</i>
INV	<i>owl:inverseOf</i>
SYM	<i>owl:SymmetricProperty</i>
TRANS	<i>owl:TransitiveProperty</i>

A Datalog rule has the form $R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ such that for each $i \in \{0, \dots, n\}$, $R_i(\bar{t}_i)$ is an atom and $\text{Var}(\bar{t}_0) \subseteq \bigcup_{1 \leq i \leq n} \text{Var}(\bar{t}_i)$. We call $R_0(\bar{t}_0)$ *head atom* and all others *body atom*. With $\text{Var}(r) \subseteq \text{VAR}$ we denote the set of all variables occurring in a Datalog rule r . A *Datalog program* is a finite set of Datalog rules. A predicate symbol occurring in the head of a rule $r \in P$ is called *intensional database predicate* (idb) for P , all other predicates are called *extensional database predicate* (edb) for P .

For any concrete given Datalog program P or Datalog query q which is applied to \mathcal{I} , we always assume that predicate and constant symbols occurring in P , and in the body of q respectively, are elements in SIG. We will rarely mention the signature since SIG is for a given database and program implicitly determined.

In our formalization, we denote the *immediate consequence operator* of a Datalog program P as T_P . T_P 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 can be non-recursively, i.e. immediately in one step, inferred from facts in \mathcal{I} under P . More formally, for each rule $r \in P$ let $q_r(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ where the $R_i(\bar{t}_i)$ are the body atoms of r . For each $R \in \text{PRED}$ let $P \upharpoonright R$ be the set of rules whose predicate symbol in the head atom is R . We set $R^{T_P(\mathcal{I})} := R^{\mathcal{I}} \cup \bigcup_{r \in P \upharpoonright R} q_r^{\mathcal{I}}$. We define $T_P^0(\mathcal{I}) := \mathcal{I}$ and $T_P(\mathcal{I})$ to be the database \mathfrak{H} where $\text{dom } \mathfrak{H} = \text{dom } \mathcal{I}$ and $R^{\mathfrak{H}} := R^{T_P(\mathcal{I})}$ for all $R \in \text{PRED}$. We set further $T_P^{n+1}(\mathcal{I}) := T_P \circ T_P^n(\mathcal{I})$.

With ω we indicate the first infinite limit ordinal and set $T_P^\omega(\mathcal{I})$ to be the database \mathfrak{H} where $\text{dom } \mathfrak{H} = \text{dom } \mathcal{I}$ and $R^{\mathfrak{H}} := \bigcup_{n < \omega} R^{T_P^n(\mathcal{I})}$ for all $R \in \text{PRED}$. With $P(\mathcal{I})$ we denote the fully materialized database of \mathcal{I} under the program P . According to [1, Chapter 12], we have $P(\mathcal{I}) = T_P^\omega(\mathcal{I})$ and in particular there is $n < \omega$ such that $R(\bar{a}) \in T_P^n(\mathcal{I})$ for every ground atom $R(\bar{a}) \in P(\mathcal{I})$.

Let V be a set of *vertices* and $E \subseteq V \times V$ an *edge relation*, then $G := (V, E)$ is called *directed graph*.

For vertices $v_0, v_1 \in V$ we call v_0 *predecessor* of v_1 , and v_1 *successor* of v_0 respectively, iff $(v_0, v_1) \in E$. A vertex that does not have a successor is called *leaf*. We define $E(v) := \{v' \in V \mid (v, v') \in E\}$ for each $v \in V$. For all $v \in V$ we define $\langle v \rangle_G$, the *v-subgraph* of G , to be the graph $G' := (V', E')$ with $E' = E \cap (V' \times V')$ where V' is the smallest set such that $v \in V'$ and whenever $v' \in V'$ then $E(v') \subseteq V'$.

A *tree* is a graph such that each vertex has exactly one predecessor, except for one, the *root*, which has no predecessor. Trees can be considered to be recursively defined, where a tree G is either a single root, i.e. $V = \{v\}$, or a tree consists of a root v and each of its successors $v' \in E(v)$ is a root of the tree $\langle v' \rangle_G$. The height of a finite tree is recursively derived as follows: if the root v is a leaf, then its height is 0, otherwise it is the maximal height of all trees $\langle v' \rangle$ plus 1, where $v' \in E(v)$. For infinite trees, the height might not be defined.

Let P be a Datalog program and \mathcal{I} a database and $R(\bar{a})$ a ground atom in $P(\mathcal{I})$. We call the pair (G, ℓ) a *Datalog proof-tree* for $R(\bar{a})$ in $P(\mathcal{I})$ if all of the following is satisfied: $G = (V, E)$ is a tree with a finite set V and $\ell : V \rightarrow \text{Atom}$ is a function, the *labeling function*, where Atom is the set of all ground atoms over SIG. Furthermore the root $v_0 \in V$ is labelled with $R(\bar{a})$, i.e. $\ell(v_0) = R(\bar{a})$, and for every leaf $v \in V$ we have $\ell(v)$ is a ground atom in \mathcal{I} . For every vertex $v \in V$ which is not a leaf there is a rule $r := R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ and an assignment $\beta : \text{Var}(r) \rightarrow \text{dom } \mathcal{I}$, such that $\ell(v) = \beta(R_0(\bar{t}_0))$ and there is a bijection $\iota : E(v) \rightarrow B$ between the successors of v and the set of body atoms $B := \{R_i(\bar{t}_i) \mid 1 \leq i \leq n\}$ of r such that $\ell(v') = \beta \circ \iota(v')$ for all $v' \in E(v)$.

A Datalog proof-tree thus represents a certain choice of rules whose application leads from atoms in \mathcal{I} to (possibly) derived atoms in $P(\mathcal{I})$. A proof by induction upon $m < \omega$ shows that every atom in $T_P^m(\mathcal{I})$ has a Datalog proof-tree of height at most m .

Finally, we define the function *lookup*: For an atomic query $Q = R(\bar{t})$ and a given database \mathcal{I} , or rather a given set of atoms \mathcal{I} , we define $\text{lookup}(R(\bar{t}), \mathcal{I})$ to be the set $\{\theta : \text{Var}(\bar{t}) \rightarrow \text{dom } \mathcal{I} \mid R(\theta(\bar{t})) \in \mathcal{I}\}$ of substitutions.

Example 3. For any ground atom $R(\bar{a})$ we have

1. $\text{lookup}(R(\bar{a}), \mathcal{I}) = \emptyset$ iff $R(\bar{a}) \notin \mathcal{I}$
2. $\text{lookup}(R(\bar{a}), \mathcal{I}) = \{\theta_\varepsilon\}$ iff $R(\bar{a}) \in \mathcal{I}$.

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

Hence $lookup(R_0(\bar{t}_0), \mathcal{J}) \bowtie lookup(R_1(\bar{t}_1), \mathcal{J})$ is the set containing all assignments $\theta : (Var(\bar{t}_0) \cup Var(\bar{t}_1)) \rightarrow dom \mathcal{J}$ which are assignments for both atoms so that $R_0(\theta(\bar{t}_0)) \in \mathcal{J}$ and $R_1(\theta(\bar{t}_1)) \in \mathcal{J}$.

3. 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 materialization, to a mere lookup in the database and is therefore faster than the latter approach, where for each answer a proof tree has to be built.

If, however, the underlying database changes frequently, then a complete materialization 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.

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 materializing as little information as necessary to yield a complete set of answers, while forward-chaining applies the rules of the given rule set until the closure is reached.

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.

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 a possible pre-materialization 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. This optimization becomes

particularly effective if patterns are pre-materialized that frequently appear during reasoning at user query time.

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

Example 4. Consider the two following rules from the OWL RL rule set:

$$\begin{aligned} T(a, p1, b) &\leftarrow T(p, SPO, p1) \wedge T(a, p, b) \\ T(x, SPO, y) &\leftarrow T(x, SPO, w) \wedge T(w, SPO, y) \end{aligned}$$

where $a, b, p, p1, x, y, w$ represent generic variables, and SPO is a constant term.

Assume we want to suppress the unfolding of all atoms of the form $T(x, SPO, y)$, modulo variable renaming. Using Datalog to implement these rules in a program, we can replace each atom by some new atom, say S , that never appears in the head of the rules. After the substitution, the previous program would become:

$$\begin{aligned} T(a, p1, b) &\leftarrow S(p, SPO, p1) \wedge T(a, p, b) \\ T(x, SPO, y) &\leftarrow S(x, SPO, w) \wedge S(w, SPO, y) \end{aligned}$$

In general, the two programs do not yield the same answers for T anymore. To restore this equality for a given database \mathcal{J} we need to calculate all " $T(x, SPO, y)$ "-triples derivable from \mathcal{J} and add them to the auxiliary relation named S in \mathcal{J} . 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 \mathcal{J} .

Note 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 $T(x, SPO, y) \leftarrow S(x, SPO, y)$.

In the following, we discuss the backward-chaining algorithm used in our approach. Then, we will show that our method to replace the original rules with others is, after a small pre-computation, harmless in the sense that everything which could be inferred under the original program can be inferred under the altered program and vice versa.

4. Hybrid reasoning: Backward-chaining

We organize this section as follows. First, we introduce in Section 4.1 the main idea behind backward-chaining, and present an overview of existing methods

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

1 from which our algorithm is derived. Then, we provide
 2 a theoretical description and analysis of our algorithm
 3 in Section 4.2.

4 4.1. Backward-chaining

7 The purpose of a backward-chaining algorithm is to
 8 derive for a given database \mathcal{J} and a Datalog program
 9 P all possible ground atoms $R(\bar{a}) \in P(\mathcal{J})$ that are
 10 answers to a given query atom Q .

11 Traditionally, users interact with RDF data sets using
 12 the SPARQL language [16] where all the triple pat-
 13 terns that constitute the body of the query are joined
 14 together according to some specific criteria. In this
 15 paper, we do not consider the problem of efficiently
 16 joining the RDF data but focus instead on the process
 17 of retrieving all triples that are needed for the query.
 18 Therefore, we target our reasoning procedure at *atomic*
 19 queries, e.g., $\top(x, \text{SCO}, y)$.

20 The algorithm that we present in the following sec-
 21 tion is a variation of the well-known algorithm QSQ
 22 (Query-subquery) [1,6,23]. The general idea behind
 23 the QSQ algorithm is to recursively rewrite the given
 24 query into many subqueries until no more rewritings
 25 can be performed and the subqueries can only be eval-
 26 uated against the knowledge base.

27 **Example 5.** To give an idea on how QSQ works, sup-
 28 pose that our initial query is

$$T(x, \text{TYPE}, \text{Person})$$

31 and that we have a generic database \mathcal{J} and the OWL
 32 RL rule set as P . Initially, the algorithm will deter-
 33 mine which rules can produce a derivation that is part
 34 of the input query. For example, it could apply the sub-
 35 class and subproperties inheritance rules (*cax-sco*
 36 and *prp-spol* in the OWL RL rule set). After it
 37 has determined them, it will move to the body of the
 38 rules and proceed evaluating them. In case these sub-
 39 queries will produce some results, the algorithm will
 40 execute the rules and return the answers to the upper
 41 level.
 42

43 With this process, the algorithm is creating a tree
 44 that has the original query as root and the rules and
 45 subqueries that might contribute to derive some an-
 46 swers as the internal nodes.

47 This tree represents all the derivation steps that are
 48 taken to derive answers of our initial query (the root)
 49 starting from some existing facts (the leaves). In Fig. 1
 50 we report an example of a part of such a tree for our
 51 query.

52 The original QSQ algorithm was introduced in
 53 1986 [24]. Unfortunately, the first version of this al-
 54 gorithm was found incomplete, but a fixed-up version
 55 was presented already the year after [6,23].

56 In principle, the QSQ algorithm extends the stan-
 57 dard SLD resolution technique [22] by applying it to
 58 a set of tuples instead of single ones [6]. An impor-
 59 tant problem of backward-chaining algorithms such as
 60 QSQ concerns the execution of recursive rules. Recur-
 61 sive rules and more in general cycles in the proof tree
 62 are an important threat to termination since they could
 63 create infinite loops in the computation.

64 To solve this problem, QSQ extends the standard
 65 SLD resolution adding two techniques: an *Admissibil-*
 66 *ity test* and a *Lemma resolution*. The resulting method,
 67 called SLD-AL, rests on the method which QSQ and
 68 all its variants are derived from. By analyzing the prop-
 69 erties of the SLD-AL resolution, the author of QSQ
 70 has proved in [23] that this algorithm always termi-
 71 nates and is complete (i.e. is able to calculate all the
 72 answers).

73 In this paper, we do not re-propose a complete de-
 74 scription of SLD-AL, and refer the reader to [23,25]
 75 for a more complete explanation. Here, we will simply
 76 sketch some characteristics of the technique which is
 77 important in our context.

78 In very general terms, SLD-AL is an abstract tech-
 79 nique to construct a computational *tree* to answer a
 80 given query. This tree, which is called *SLD tree*, resem-
 81 bles the tree in Fig. 1, and is explored by an algorithm
 82 such as QSQ in order to find all the answers (called
 83 atomic lemmas) that satisfy the input query. It is cru-
 84 cial that the SLD tree is *finite* and *complete*, otherwise,
 85 it would be impossible for an algorithm to terminate
 86 and compute all the derivations. These properties do
 87 hold for SLD-AL trees, as shown in [25].

88 The main idea behind the *AL* technique is the fol-
 89 lowing: when the algorithm computes a tree and needs
 90 to evaluate a new query, it applies an “admissibility
 91 test”, to verify whether this query can be resolved us-
 92 ing the rules. If the new query is “similar” to a previous
 93 one, then the query is evaluated using only the answers
 94 produced so far (“lemma resolution”). In this way, the
 95 program does not get trapped in a loop generating an
 96 infinite tree.

97 The SLD tree can be explored in several ways. In
 98 the general case, the search strategy can determine the
 99 completeness of the approach, but in Datalog any strat-
 100 egy is equivalent. For example, the tree can be con-
 101 structed using an iterative depth-first strategy as pro-
 102 posed with the original QSQ algorithm in [24], or

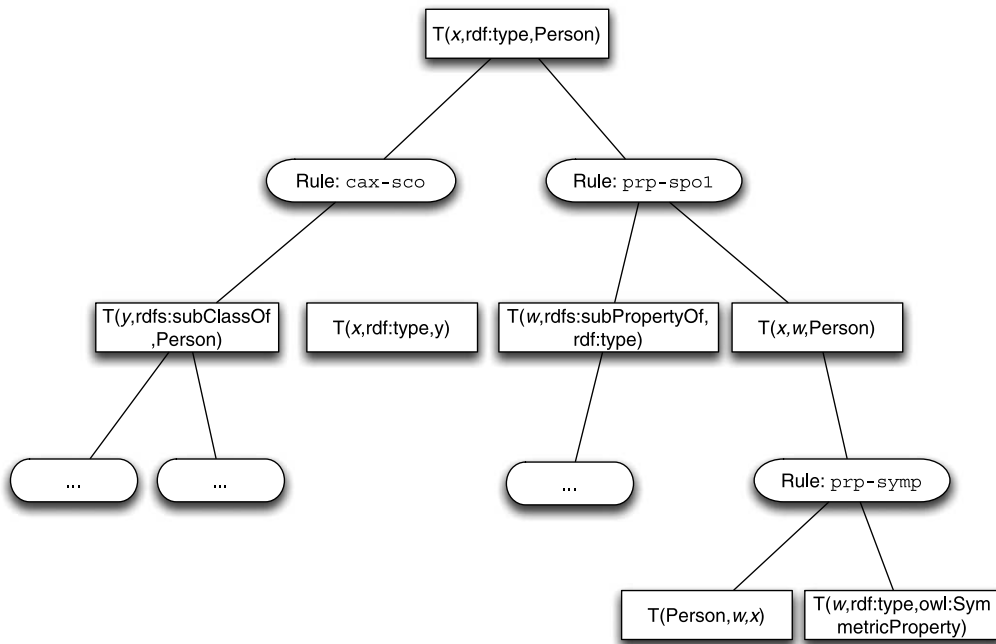


Fig. 1. Example of the execution of backward-chaining for the input query $T(x, \text{TYPE}, \text{Person})$ and the OWL RL rules.

using a more sophisticated constructive search as in QoSaq (QSQ+gloBAloptimization) [25].

In the early research on these methods, much emphasis was put on optimizing the computation to avoid redundancy. For example, the original QSQ algorithm traverses the tree using a depth-first strategy, and repeats the query execution at every node until it retrieves all answers for that subgoal [6]. In this way, it is able to cache the results for that subquery and reuse them during the evaluation of the rest of the tree. QoSaq [25] goes further than the original QSQ algorithm proposing one more solution to this problem: here, a copy of the tree is maintained in main memory and a “waking” mechanism is used to feed new answers derived in one node to other equivalent queries that appear in other branches of the tree.

These techniques are very efficient in optimizing the resolution process, but have the drawback that they are very difficult to be implemented in a parallel (and possibly distributed) environment. In fact, the original depth-first strategy used by QSQ requires a sequential search of the tree, otherwise expensive synchronization mechanisms must be used. The “waking” mechanism proposed by QoSaq cannot be applied in a distributed environment, since it requires to maintain a copy of the tree in main memory and this mechanism would be slowed down by a network access.

Therefore, we adapted the original QSQ algorithm so that it can be more easily parallelized paying the price of duplicate derivation and possibly redundant computation. We will present the algorithm in the following section and show that the properties of termination soundness, and completeness are still valid.

4.2. Our approach

We introduce two key differences from the original QSQ algorithm, which aim is to improve the parallelization of the computation:

- Unlike QSQ, our algorithm does not construct the proof-tree sequentially but in parallel by applying the rules on separate threads and in an asynchronous manner. For example, if we look back at Fig. 1, the execution of rules *cax-sco* and *prp-spo1* is performed concurrently by different threads. This execution strategy makes the implementation and the maintenance of the global data structure, used for caching results of previous queries, difficult and inefficient. We hence choose to replace this mechanism with one that only remembers which queries were already executed along single paths of the tree. While such a choice might lead to some duplicate answers because the same queries can be repeated multiple

Algorithm 1 (Backward-chaining algorithm). \mathcal{I} and P are global constants, where \mathcal{I} is a finite set of facts and P is a Datalog program. Tmp and Mat are global variables, where Mat stores results of previous materialization rounds and Tmp stores the results of the current round. The parameter Q represents an atomic query, i.e. an atom. Both functions *main* and *infer* return a set of atoms, while the function *lookup* returns a set of substitutions. We say $Q \in PrevQueries$ (cf. line 15) iff there is $Q' \in PrevQueries$ s.t. $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$.

```

1  function main(Q)
2    New, Tmp, Mat :=  $\emptyset$ 
3    repeat
4      Mat := Mat  $\cup$  New  $\cup$  Tmp
5      New := infer(Q,  $\emptyset$ )
6    until New  $\cup$  Tmp  $\subseteq$  Mat  $\cup$   $\mathcal{I}$ 
7    return New
8  end function
9
10 function infer(Q, PrevQueries)
11
12  //This cycle is executed in parallel
13  all_subst := lookup(Q,  $\mathcal{I} \cup Mat$ )
14  for  $(\forall r \in P$  s.t.  $Q$  is unifiable
15    with  $r$ .HEAD and  $Q \notin PrevQueries$ )
16     $\theta_h :=$  MGU( $Q, r$ .HEAD)
17    subst :=  $\{\theta_h\}$ 
18    for  $\forall p \in r$ .BODY
19      tuples := infer( $\theta_h(p), PrevQueries \cup Q$ )
20      Tmp := Tmp  $\cup$  tuples
21      subst := subst  $\bowtie$  lookup( $\theta_h(p), tuples$ )
22    end for
23  all_subst := all_subst  $\cup$  (subst  $\circ \theta_h$ )
24  end for
25
26  return  $\bigcup_{\theta \in all\_subst} \{\theta(Q)\}$ 
27
28 end function

```

times in different parts of the proof tree, it allows the computation to be performed in parallel limiting the usage of expensive synchronization mechanisms;

- Because the proof tree is built in parallel, repeating every query until no new results are found is an inefficient operation: the same query can appear multiple times in 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 that stores during every iteration all the intermediate derivations.

We report the algorithm using pseudocode in Algorithm 1. The procedure *main* is the main function used to invoke the backward-chaining procedure for a given atomic query Q . *main* 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 intermedi-

ate answers that were inferred in the process and 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 \mathcal{I} . If no new tuple was 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).

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 applied to derive new answers for Q (lines 14–15) and calculates the substitution θ_h to unify the head of the applicable rule with the query Q (line 16).

It proceeds with evaluating the body of the rule (lines 16–23) storing in *tuples* and *Tmp* the retrieved answers (lines 19–20), and performing the joins necessary according to the rule body (line 21).

In line 23, each assignment in *subst* is composed with the unifier under which it has been derived in the for-loop (line 18–22), which renders it into an assignment for Q . All these assignments are eventually copied into *all_subst* from which a set of answers for Q is derived (line 26) The answers are then returned to the function caller. After the whole recursion tree has been explored exhaustively, *infer* returns control to the function *main*, where the derived answers are copied into the variable *New*. The process is repeated until the closure is reached.

To facilitate the understanding of this algorithm and more in particular of the function *infer*, consider the following example:

Example 6. Suppose that we have a program P that consists of a single rule (notice that x, y, z are variables)

$$r_0 := T(x, \text{TYPE}, y) \leftarrow T(z, \text{SCO}, y) \wedge T(x, \text{TYPE}, z)$$

the input query is $Q := (a, \text{TYPE}, u)$, containing only one variable, u , and \mathcal{I} contains solely the ground atoms $T(a, \text{TYPE}, c)$ and $T(c, \text{SCO}, d)$.

The call *main*(Q) executes *infer*(Q, \emptyset) which performs a lookup in line 13, setting *all_subst* = $\{\{u/c\}\}$. It is then checked for all rules r whether the head of r is unifiable with Q (lines 14–15). In our example, only rule r_0 satisfies this condition, and the algorithm computes some MGU θ_h (line 16), e.g. $\theta_h := \{x/a, y/u\}$.

θ_h is applied to each body atom of r_0 and *infer* is recursively invoked. In our example, the first recursive

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

1 call would be $\text{infer}(T(z, \text{SCO}, u), \{Q\})$. We obtain at
 2 first the substitution $\{z/c, u/d\}$ in line 13 which is
 3 stored in the local variable all_subst . Since no rule
 4 head unifies with $T(z, \text{SCO}, u)$ the program jumps to
 5 line 26 and returns $\{\{z/c, u/d\}(T(z, \text{SCO}, u))\} =$
 6 $\{T(c, \text{SCO}, d)\}$. Hence the join in line 21 evaluates
 7 to $\{\{z/c, u/d\}\}$ and $\text{subst} = \{\{z/c, u/d\}\}$ after
 8 line 21.

9 The inner for-loop (lines 18–22) moves to the
 10 next predicate and launches the second recursive call
 11 $\text{infer}(T(a, \text{TYPE}, z), \{Q\})$. Line 13 sets all_subst to
 12 $\{z/c\}$ and forgoes lines 14–24 since $T(a, \text{TYPE}, z)$
 13 equals Q up to variable renaming and $\text{PrevQueries} =$
 14 $\{Q\}$.

15 Returning $\{T(a, \text{TYPE}, c)\}$ to the computation
 16 one recursion level above, the result of the join
 17 $\{\{z/c, u/d\}\} \bowtie \{\{z/c\}\} = \{\{z/c, u/d\}\}$ in line
 18 21 will be stored in subst . Line 23 sets all_subst to
 19 $\{\{u/c\}, \{z/c, u/d, x/a, y/d\}\}$ which is the result of
 20 $\{\{u/c\}\} \cup \{\{z/c, u/d\}\} \circ \{x/a, y/u\}$.

21 Each substitution in all_subst is applied to Q in line
 22 26 and $\text{Result} := \{T(a, \text{TYPE}, c), T(a, \text{TYPE}, d)\}$ is
 23 returned to the function main where they are stored
 24 in Mat . The function main will repeat the computa-
 25 tion once again to ensure that no more triples can
 26 be derived. In this last loop-pass $\text{infer}(Q, \emptyset)$ returns
 27 the atoms in Result which it obtains this time from a
 28 lookup in line 13. Result is stored in New which is then
 29 returned to the user (line 7).

30 By definition, unifiers and therefore the MGU rely
 31 on variable disjoint atoms. We will show for the con-
 32 struction of the MGU in line 16 that we may w.l.o.g.
 33 assume that every query Q' occurring in the compu-
 34 tation of $\text{infer}(Q, \emptyset)$ is variable disjoint to every rule
 35 $r \in P$. In fact, we assume that the for-loop in Algo-
 36 rithm 1 line 14 iterates over variable renamed versions
 37 of rules $r \in P$ which are variable disjoint to the given
 38 Q' .

39 To this end, we show that for every Datalog program
 40 P and atomic query Q a finite variable set V suffices
 41 such that for every query Q' occurring in the compu-
 42 tation of $\text{infer}(Q, \emptyset)$ and every rule $r \in P$ a variable
 43 renaming ρ exists such that the variables in Q' and the
 44 renamed variables in r are disjoint, i.e.

$$45 \quad \text{Var}(Q') \cap \rho \circ \text{Var}(r) = \emptyset \text{ and } \rho \circ \text{Var}(r) \subseteq V.$$

46
 47 To this end let \mathcal{A} comprise Q and all atoms occur-
 48 ring in P . We set V_0 to be the set of all variables in \mathcal{A}
 49 and $V := V_0 \cup V_1$ where V_1 is a disjoint copy of V_0 . We

52 define \mathcal{Q} to be the closure of \mathcal{A} under all substitutions
 53 $\theta : V \rightarrow (V \cup \text{CONS})$ where CONS are the constants
 54 occurring in \mathcal{A} . Obviously, \mathcal{Q} is finite.

55 Lemma 1.

- 56 1. For every $Q' \in \mathcal{Q}$ and every $r \in P$ there is a
 57 variable renaming $\rho : \text{Var}(r) \rightarrow V \setminus \text{Var}(Q')$.
 58
- 59 2. We have $Q' \in \mathcal{Q}$ for every subsequent procedure
 60 call $\text{infer}(Q', \text{PrevQueries})$ of $\text{infer}(Q, \emptyset)$.
 61

62 Proof.

- 63 1. For a set M we denote with $|M|$ its cardinality.
 64 Let $Q' \in \mathcal{Q}$ and $r \in P$ be arbitrary. For every
 65 atom $R(\bar{t}) \in \mathcal{A}$ we know $|\text{Var}(R(\bar{t}))| \leq \frac{1}{2}|V|$.
 66 Since Q' is the result of a variable substitution
 67 within an atom in \mathcal{A} , we have $|\text{Var}(Q')| \leq \frac{1}{2}|V|$
 68 and hence that $|V \setminus \text{Var}(Q')| \geq \frac{1}{2}|V|$. Similarly
 69 we know that $|\text{Var}(r)| \leq \frac{1}{2}|V|$. Hence there is an
 70 injective substitution $\rho : \text{Var}(r) \rightarrow V \setminus \text{Var}(Q')$.
 71
- 72 2. The proof is carried out via the nesting depth $k <$
 73 ω of the procedure calls. If $k = 0$, the procedure
 74 call is $\text{infer}(Q, \emptyset)$ and since $Q \in \mathcal{Q}$ the claim is
 75 true.

76 Assume we are in a subsequent procedure call
 77 $\text{infer}(Q', \text{PrevQueries})$ in nesting depth $k < \omega$.
 78 The induction hypothesis yields that $Q' \in \mathcal{Q}$. For
 79 every rule $r \in P$ used in a loop pass (line 14–24),
 80 item 1. yields a variable renaming $\rho : \text{Var}(r) \rightarrow$
 81 $V \setminus \text{Var}(Q')$. Let the variables of r be renamed
 82 by ρ , then $\text{Var}(\text{MGU}(Q', r.\text{HEAD})(p)) \subseteq V$ for
 83 all body atoms p of r . The argument Q'' in sub-
 84 sequent procedure calls $\text{infer}(Q'', \text{PrevQueries} \cup$
 85 $\{Q'\})$ (line 19) which have nesting depth $k + 1$
 86 is of the form $Q'' := \text{MGU}(Q', r.\text{HEAD})(p)$ and
 87 hence $Q'' \in \mathcal{Q}$. \square
 88

89 We will now discuss the correctness of our algo-
 90 rithm in three steps, which are termination, soundness
 91 and completeness. We shall first show termination and
 92 soundness, for which we will furnish two further lem-
 93 mas: The first, Lemma 2, conceptualizes the function
 94 calls executed during $\text{infer}(Q, \emptyset)$ as a tree, where ele-
 95 ments are connect by edges of the “calls”-relation, and
 96 shows that this tree has finite depth and thus yields an
 97 argument towards termination. The second, Lemma 3,
 98 shows that all calls in this tree yield only results in
 99 $P(\mathcal{I})$, i.e. Lemma 3 provides a soundness argument.

100 Consider database \mathcal{I} , a Datalog program P and an
 101 atom Q . Let \mathcal{Q} be the set of atoms defined above
 102 Lemma 1.

1 **Lemma 2.** Each call $\text{infer}(Q, \emptyset)$ entails at most
2 finitely many recursive calls to infer .

3
4 *Proof.* An inductive argument over the nesting depth
5 of procedure calls shows that in the n -th nested pro-
6 cedure call, PrevQueries contains n elements. Since
7 $\text{PrevQueries} \subseteq \mathcal{Q}$ (cf. Lemma 1 item 2.) and both
8 for-loops iterate over finite sets, infer is called at most
9 finitely many times. \square

10
11 With \supseteq (being a proper super-set) we obtain a well-
12 founded order on the powerset of \mathcal{Q} which we shall use
13 for further inductive arguments.

14 **Lemma 3.** Let $P(\mathcal{J})$ be the materialization of \mathcal{J} under
15 P . If $\text{Mat}, \text{Tmp} \subseteq P(\mathcal{J})$ then for all $Q \in \mathcal{Q}$ and all
16 subsets $\text{PrevQueries} \subseteq \mathcal{Q}$ we have that

$$\text{infer}(Q, \text{PrevQueries}) \subseteq P(\mathcal{J})$$

17
18
19
20 and $\text{Tmp}' \subseteq P(\mathcal{J})$ where Tmp' is Tmp after the execu-
21 tion of $\text{infer}(Q, \text{PrevQueries})$.

22
23 *Proof.* The proof is carried out by induction upon
24 PrevQueries in the powerset of \mathcal{Q} ordered by \supseteq . For
25 the base case let $Q \in \mathcal{Q}$ be arbitrary and execute
26 $\text{infer}(Q, \mathcal{Q})$. Then $\text{lookup}(Q, \mathcal{J} \cup \text{Mat})$ in line 13 re-
27 turns a set of assignments $\beta : \text{Var}(Q) \rightarrow \text{dom } \mathcal{J}$,
28 such that $\beta(Q) \in (\mathcal{J} \cup \text{Mat})$. Since $Q \in \mathcal{Q}$, lines 14–24
29 are skipped and hence

$$\text{infer}(Q, \mathcal{Q}) \subseteq P(\mathcal{J})$$

30
31
32
33 and $\text{Tmp}' = \text{Tmp} \subseteq P(\mathcal{J})$.

34 For the induction step, let $\text{PrevQueries} \subseteq \mathcal{Q}$ be ar-
35 bitrary and assume as induction hypothesis that for all
36 \mathcal{R} with $\text{PrevQueries} \subsetneq \mathcal{R} \subseteq \mathcal{Q}$ and for all $Q \in \mathcal{Q}$
37 we have that if $\text{Mat}, \text{Tmp} \subseteq P(\mathcal{J})$ then $\text{infer}(Q, \mathcal{R}) \subseteq$
38 $P(\mathcal{J})$ and $\text{Tmp}' \subseteq P(\mathcal{J})$ where Tmp' is the updated set
39 Tmp after the procedure call $\text{infer}(Q, \mathcal{R})$.

40 Execute $\text{infer}(Q, \text{PrevQueries})$: As in the induction
41 base, all_subst contains after line 13 only assignments
42 β such that $\beta(Q) \in (\mathcal{J} \cup \text{Mat})$. If $Q \in \text{PrevQueries}$ up
43 to variable renaming, we skip lines 14–24 ending up
44 with the same outcome as in the induction base.

45 Hence assume $Q \notin \text{PrevQueries}$ modulo variable
46 renamings. In each loop-pass of the outer-loop and
47 inner-loop, the induction hypothesis yields that

$$\text{infer}(\theta_h(p), \text{PrevQueries} \cup \{Q\}) \subseteq P(\mathcal{J}).$$

48
49
50
51 Hence $\text{Tmp}' \subseteq P(\mathcal{J})$.

52 Similar to line 13, $\text{lookup}(\theta_h(p), \text{tuples})$ contains
53 exactly those assignments $\beta : \text{Var}(\theta_h(p)) \rightarrow \text{dom } \mathcal{J}$
54 such that $\beta \circ \theta_h(p) \in \text{tuples}$. Hence every $\beta \in \text{subst}$
55 after the join \bowtie has been performed (line 22), satis-
56 fies $\beta \circ \theta_h(p) \in \text{tuples}$ for all $p \in r.BODY$. Datalog
57 requires all variables of the head to be covered
58 by some atom in the body, so we know that each
59 $\beta \in \text{subst}$ is an assignment for $\theta_h(r.HEAD)$ where
60 $\beta \circ \theta_h(r.HEAD) \in P(\mathcal{J})$. Additionally, since θ_h is
61 the unifier for Q and $r.HEAD$, we know that every
62 $\beta \circ \theta_h$ in all_subst is an assignment for Q . Hence, we
63 have $\theta(Q) \in P(\mathcal{J})$ for all $\theta \in \text{all_subst}$ which shows
64 $\text{infer}(Q, \text{PrevQueries}) \subseteq P(\mathcal{J})$. \square

4.2.1. Termination

65
66 We first concentrate on the termination of the pro-
67 cedure call $\text{infer}(Q, \emptyset)$ in the function main . Let \mathcal{J} be
68 a database over a finite signature SIG: by definition
69 $\text{dom } \mathcal{J}$ is finite. Lemma 3 yields that in every repeat-
70 loop pass (lines 3–6)

$$\text{New}, \text{Tmp}, \text{Mat} \subseteq P(\mathcal{J}) \quad (*)$$

71
72
73
74 where $P(\mathcal{J})$ is the materialization of \mathcal{J} under P . An
75 inductive argument over $\text{PrevQueries} \subseteq \mathcal{Q}$ shows that
76 under the precondition $(*)$ for all $Q \in \mathcal{Q}$ every proce-
77 dure call $\text{infer}(Q, \text{PrevQueries})$ terminates: Lemma 2
78 shows that there are only finitely many calls to infer ,
79 but we can now show that also every call to lookup in
80 line 13 and line 21 yields a finite result (via some finite
81 computation) and thus \bowtie in line 21 terminates.

82
83 In every repeat-loop pass, Tmp or New grow or the
84 loop is terminated. Since Tmp and New are bounded
85 by $P(\mathcal{J})$, which is finite, the repeat-loop terminates
86 after finitely many passes. This shows that for every
87 database \mathcal{J} , every Datalog program P and every atomic
88 query Q the function $\text{main}(Q)$ terminates.

4.2.2. Soundness

89
90 The return value of $\text{main}(Q)$ is the result of the call
91 $\text{infer}(Q, \emptyset)$ in the last repeat loop pass (cf. line 5 in
92 Algorithm 1). Hence, in order to show soundness, we
93 have to show for all repeat-loop passes that the re-
94 turn value of $\text{infer}(Q, \emptyset)$ only contains answers to Q
95 from $P(\mathcal{J})$. To this end, assume \mathcal{J} is a database, P is
96 a Datalog program and Q is an atomic query. In ev-
97 ery repeat-loop pass, $\text{infer}(Q, \emptyset)$ contains only ground
98 atoms from $P(\mathcal{J})$ that unify with Q : Using Lemma 3,
99 we know that $\text{Tmp}, \text{Mat} \subseteq P(\mathcal{J})$ for every repeat-loop
100 pass (lines 3–6). Thus, line 13 only yields assignments
101 β such that $\beta(Q) \in (\mathcal{J} \cup \text{Mat})$ and hence such that
102 $\beta(Q) \in P(\mathcal{J})$. Using Lemma 3 again on line 19, we

1 know that $\beta \circ \theta_h(p) \in P(\mathfrak{I})$ for every assignment
 2 $\beta \in \text{lookup}(\theta_h(p), \text{tuples})$ in line 21 and every body
 3 atom p of a rule whose head unifies with Q . Hence
 4 $\beta \in \text{subst}$ after line 22 is an assignment such that
 5 $\beta \circ \theta_h(Q) \in P(\mathfrak{I})$. Hence the returned set in line 26
 6 only contains ground atoms which are answers to Q
 7 and are in $P(\mathfrak{I})$.

8 4.2.3. Completeness

9 Let \mathfrak{I} be a given database, P a given Datalog pro-
 10 gram and $Q := R'(\bar{t})$ an atomic query. To show the
 11 completeness of Algorithm 1, we need to prove that
 12 every atom in $R^{P(\mathfrak{I})}$ which unifies with Q can be de-
 13 rived by $\text{main}(Q)$. This claim is shown via Proposition
 14 1 below, as the latter holds in particular for all answers
 15 to the input query Q derived under P from \mathfrak{I} . Propo-
 16 sition 1 however rests on Lemma 4. The lemma states
 17 that if a ground atom $R(\bar{a})$ appears as label in a Data-
 18 log proof-tree for some answer $R'(\bar{b})$ to Q , then $R(\bar{a})$
 19 is an answer to some query Q_n which will occur dur-
 20 ing a computation of $\text{infer}(Q, \emptyset)$. Note that it does not
 21 show that $\text{infer}(Q, \emptyset)$ derives $R(\bar{a})$.

22 We call an atomic query Q *blocked* in the pro-
 23 cedure call $\text{infer}(Q, \text{PrevQueries})$ if there is $Q' \in$
 24 PrevQueries such that $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$.

25 **Lemma 4.** *Let $Q := R'(\bar{t})$ be an atomic query and*
 26 *$R(\bar{a})$ a label, which appears in a Datalog proof-tree*
 27 *(G, ℓ) for some answer to Q in $P(\mathfrak{I})$. Then there is*
 28 *a subsequent procedure call $\text{infer}(Q_n, \text{PrevQueries})$*
 29 *of $\text{infer}(Q, \emptyset)$ such that Q_n is a non-blocked atomic*
 30 *query and $R(\bar{a})$ is an answer to Q_n derived under P*
 31 *in \mathfrak{I} .*

32 *Proof.* Let $R'(\bar{b})$ be an answer to Q which has a Data-
 33 log proof-tree (G, ℓ) in $P(\mathfrak{I})$ containing a label $R(\bar{a})$.
 34 We have to show, that there is a sequence of atomic
 35 queries Q_0, \dots, Q_n such that

- 36 1. $Q = Q_0$ and $R(\bar{a})$ unifies with Q_n
- 37 2. for each $i \in \{0, \dots, n\}$ there is a rule $r \in P$ and
 38 $\theta := \text{MGU}(Q_i, r.\text{HEAD})$ such that $Q_{i+1} = \theta(p)$,
 39 where p is some body-atom of r
- 40 3. no query is blocked, i.e. there is no subsequence
 41 $Q_i \dots Q_k$ with $0 \leq i < k \leq n$ such that Q_i is up
 42 to variable renaming equal to Q_k ($Q_i \sqsubseteq Q_k$ and
 43 $Q_k \sqsubseteq Q_i$).

44 3. guarantees in particular that the condition $Q \notin$
 45 PrevQueries in line 15 is true when the procedure call
 46 $\text{infer}(Q_i, \{Q_0, \dots, Q_{i-1}\})$ is executed. Hence, if 1–3

hold, $\text{infer}(Q, \emptyset)$ will, according to lines 14–24 of Al-
 52 gorithm 1, call in ascending sequence

$$53 \text{infer}(Q_i, \{Q_0, \dots, Q_{i-1}\})$$

54 where $0 \leq i \leq n$.

55 In a first step we specify a sequence of rules
 56 r_0, \dots, r_n which leads from $R'(\bar{b})$ to the atom $R(\bar{a})$
 57 and then we determine a sequence of queries
 58 Q_0, \dots, Q_n .

59 Let $R'(\bar{b})$ be the atom which unifies with the in-
 60 put query Q and in whose Datalog proof-tree (G, ℓ)
 61 $R(\bar{a})$ appears. Then either (G, ℓ) has height 0 and thus
 62 $R'(\bar{b}) = R(\bar{a})$ and $Q_n := Q$ is found, or there is
 63 a sequence of rule applications r_0, \dots, r_n such that
 64 $R'(\bar{b})$ unifies with the head of r_0 via some MGU θ_0
 65 and for all $i \in \{0, \dots, n-1\}$ some unified body-atom
 66 $B_{i,k_i} = \theta_i(R_{i,k_i}(\bar{t}_{i,k_i}))$ of r_i unifies via some MGU
 67 θ_{i+1} with the head of r_{i+1} and finally $R(\bar{a})$ unifies
 68 with some unified body-atom $B_{n,k_n} = \theta_n(R_n(\bar{t}_{n,k_n}))$
 69 of r_n .

70 Fix this sequence of rules r_0, \dots, r_n . Since $R'(\bar{b})$
 71 is an answer to Q (i.e. a ground atom) and unifies
 72 with the head atom H_0 of r_0 , Q unifies with H_0 yield-
 73 ing $\vartheta_0 := \text{MGU}(Q, H_0)$. For all $i \in \{0, \dots, n-1\}$
 74 the unified body-atom $Q_i := \vartheta_i(R_{i,k_i}(\bar{t}_{i,k_i}))$
 75 of r_i unifies with the head H_{i+1} of r_{i+1} yielding
 76 $\vartheta_{i+1} := \text{MGU}(Q_i, H_{i+1})$, so that we finally reach
 77 the body atom $R_{n,k_n}(\bar{t}_{n,k_n})$ of r_n where $Q_n :=$
 78 $\vartheta_n(R_{n,k_n}(\bar{t}_{n,k_n}))$ is the query which unifies with
 79 $R(\bar{a})$.

80 We hence obtain a sequence Q_0, \dots, Q_n satisfy-
 81 ing items 1 and 2. We shall show that for every se-
 82 quence satisfying items 1 and 2 there is a sequence
 83 Q'_0, \dots, Q'_m satisfying items 1–2 and 3:

84 The claim is clear, if the sequence is of length 1:
 85 Q_0 is never blocked. Let $Q_0 \dots Q_n$ be a sequence
 86 of length $n+1$ where Q_i equals Q_k up to vari-
 87 able renaming and $0 \leq i < k \leq n$. Then the
 88 head of r_{k+1} unifies with the query Q_i . The se-
 89 quence $Q_0, \dots, Q_i, Q_{k+1} \dots Q_n$ is properly shorter
 90 than $Q_0 \dots Q_n$ and satisfies items 1–2. The induction
 91 hypothesis yields a sequence Q'_0, \dots, Q'_m which sat-
 92 isfies items 1–3. \square

93 **Proposition 1.** *Let P, \mathfrak{I} and Q be fixed and \mathcal{Q} as*
 94 *defined above Lemma 1. Let $R'(\bar{b})$ be an answer to*
 95 *Q which has a Datalog proof-tree (G, ℓ) in $P(\mathfrak{I})$*
 96 *containing a label $R(\bar{a})$. Then there is a repeat-loop*
 97 *pass in $\text{main}(Q)$ from which onward for every atomic*
 98 *query $Q' \in \mathcal{Q}$ which unifies with $R(\bar{a})$ and for all*
 99 *100*
 101 *102*

1 $PrevQueries \subseteq \mathcal{Q}$ every call $infer(Q', PrevQueries)$
 2 returns $R(\bar{a})$.

3
 4 *Proof.* Let $R'(\bar{b})$ be an answer to Q which has a Datalog proof-tree (G, ℓ) in $P(\mathcal{J})$ containing a label $R(\bar{a})$. We prove by induction upon $k < \omega$, that the atom $R(\bar{a})$ is yielded after the k -th repeat-loop pass by every call $infer(Q', PrevQueries)$, if Q' unifies with $R(\bar{a})$ and the minimal height of some Datalog proof tree for $R(\bar{a})$ is equal to k .

5
 6
 7
 8
 9
 10
 11 If there is a Datalog proof tree for $R(\bar{a})$ of height 0 and Q' unifies with $R(\bar{a})$, then $infer(Q', PrevQueries)$ will produce $R(\bar{a})$ for all $PrevQueries \subseteq \mathcal{Q}$ in the look-up of line 13 which will be returned (cf. line 26) for all further repeat-loop passes.

12
 13
 14
 15
 16 Assume there is a Datalog proof tree (G', ℓ') for $R(\bar{a})$ of height $k + 1$ and we have started the $k + 1$ repeat-loop pass. Lemma 4 shows that there is subsequent procedure call $infer(Q_n, PrevQueries)$ of $infer(Q, \emptyset)$ such that Q_n is an unblocked atomic query, i.e. there is no $Q' \in PrevQueries$ with $Q_n \sqsubseteq Q'$ and $Q' \sqsubseteq Q_n$, and $R(\bar{a})$ unifies with Q_n .

17
 18
 19
 20
 21
 22
 23 Since (G', ℓ') is of height $k + 1$ there is a rule $r : R(\bar{t}) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_m(\bar{t}_m)$ and a variable assignment β such that $R(\beta(\bar{t})) = R(\bar{a})$ and for each $i \in \{1, \dots, m\}$ the fact $R_i(\beta(\bar{t}_i))$ has a proof tree of height at most k under P in \mathcal{J} .

24
 25
 26
 27
 28
 29 The head H of r and Q_n unify with the ground atom $R(\bar{a})$, so there is $\theta := MGU(Q_n, H)$ and each $R_i(\beta(\bar{t}_i))$ unifies with $Q'_i := R_i(\theta(\bar{t}_i))$ with $i \in \{1, \dots, m\}$. Since Q_n is not blocked, the subsequent procedure call $infer(Q'_i, PrevQueries \cup \{Q_n\})$ for all $i \in \{1, \dots, m\}$ is issued.

30
 31
 32
 33
 34
 35 By the induction hypothesis, for all $i \in \{1, \dots, m\}$, $infer(Q'_i, PrevQueries \cup \{Q_n\})$ yields $R_i(\beta(\bar{t}_i))$. Hence $R(\bar{a})$ is returned by $infer(Q_n, PrevQueries)$ at the very latest in the $n + 1$ repeat-loop pass and eventually added to Mat (cf. line 4) so that after the $n + 1$ repeat-loop pass for every $PrevQueries \subseteq \mathcal{Q}$ every subsequent call $infer(Q', PrevQueries)$ that unifies with $R(\bar{a})$ will return this fact as look-up in line 13. \square

36
 37
 38
 39
 40
 41
 42
 43
 44 By proving Proposition 1, we have shown that Algorithm 1 is complete: Every answer to a given query Q has a Datalog proof-tree and is the label of the root of this proof-tree. Proposition 1 shows that this answer is eventually derived by $infer(Q, \emptyset)$ (line 5) and thus returned by $main(Q)$ (line 7). This completes the three steps to prove correctness w.r.t. to retrieval of exactly those answers to Q under $P(\mathcal{J})$.

5. Hybrid reasoning: Pre-materialization

52
 53
 54 So far, we have made no difference in the description of our backward-chaining algorithm between sub-queries that are pre-computed and those which are not. However, the pre-materialization of a selection of queries allows us to substantially improve the implementation and performance of backward-chaining by exploiting the fact that these queries can be retrieved with a single lookup.

55
 56
 57
 58
 59
 60
 61 In our implementation, the results of these queries are stored in main memory so that the joins required by the rules can be more efficiently executed. Also, the availability of the pre-materialized queries in memory allows us to implement another efficient information passing strategy to reduce the size of the proof tree by identifying beforehand whether a rule can contribute to deriving answers for a given query.

62
 63
 64
 65
 66
 67
 68
 69 In fact, the pre-materialization can be used to determine early failures: Emptyness 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 answers, 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.

70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80 To better illustrate this concept, we proceed with an example. Suppose we have the proof tree described in Fig. 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-spol`.

81
 82
 83
 84
 85
 86
 87 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 $T(w, SPO, TYPE)$ will also be the subject of $T(w, TYPE, SYM)$. If both patterns are pre-computed, then we know beforehand all the possible ‘ w ’, and therefore we can immediately perform an intersection between the two sets. If the intersection is non-empty, the reasoner proceeds executing rule `prp-symp`, otherwise it can skip its execution since the rule will never fire.

88
 89
 90
 91
 92
 93
 94
 95
 96
 97 It is very unlikely that the same property appears in all the terminological patterns, therefore an information passing strategy that is based on the pre-materialized triple patterns is very effective in significantly reducing the tree size and hence improving performance.

52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102

Furthermore, our implementation of this algorithm applies a *sideways information passing strategy* [2,3] to improve the execution of the joins required by the rules. This technique consists of “passing” admissible values to the variables of the following queries in order to limit the retrieval to only facts that can contribute to the join.

To illustrate this concept, consider Example 6: Here, when the algorithm needs to invoke the function *infer* with the atomic query $T(a, \text{TYPE}, u)$, even though the variable u could in principle assume any value, in practice the implementation knows already that only the values of u in *subst* are admissible (in our case c), because only those can lead to a successful join. Thus, the implementation can link these values to the variable u so that in a subsequent call of, for example, *lookup*, the computation is limited to retrieve only these values and not all possible u . This technique of passing values between the queries is well-known in rule-based systems, and applied in nearly all implementations.

The hybrid approach thus consists of materializing parts of the knowledge base beforehand and then using on-demand querying techniques to answer user queries. To this end we pre-materialize certain atomic queries and store their results in new *edb* relations before the user can query the knowledge base. The corresponding atoms in the original rules are replaced accordingly and this new rule set is then used to infer the answers to the user query.

In order to prove the correctness of our method for hybrid reasoning, we proceed as follows: First, in Section 5.1, we formalize and discuss the completeness of the pre-materialization algorithm. Next, in Section 5.2, we show that replacing the pre-materialized body atoms in the original rules with atoms using the introduced *edbs* is harmless (after the pre-materialization algorithm is computed), since this modified program computes the same answers as the original one. This last argument finally settles the correctness of our entire approach, since it ensures that, after the pre-materialization is computed, no derivation will be missed at query time.

5.1. Pre-materialization

Let \mathcal{J} be a database and P the program with a set 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{J} for each $Q \in L$ and populate these by setting $S_Q^{\mathcal{J}} := \text{main}(Q)$

Algorithm 2 (Overall algorithm of the pre-computation procedure). L is a set containing all queries that were selected for pre-materialization, $Ruleset$ is a constant containing a program P and $Database$ represents \mathcal{J} .

```

1  for every  $Q \in L$ 
2    introduce a new predicate symbol  $S_Q$  to
      Database
3  end for
4
5  for every rule  $p : R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$  in
      Ruleset
6    for every  $Q \in L$  and  $i \in \{1, \dots, n\}$ 
7      if  $R_i(\bar{t}_i) \sqsubseteq Q$  then
8        replace  $R_i(\bar{t}_i)$  in  $p$  with  $S_Q(\bar{t}_i)$ 
9      end if
10   end for
11   add this (altered) rule to NewRuleset
12 end for
13
14 Derivation :=  $\emptyset$ 
15 repeat
16   Database := Database  $\cup$  Derivation
17   for every  $R(\bar{t}) \in L$ 
18     Perform  $S_{R(\bar{t})}(\bar{t}) \leftarrow R(\bar{t})$  on Database
19   end for
20
21   for every  $Q$  in  $L$ 
22     Derivation := Derivation  $\cup$  main( $Q$ ) using
      NewRuleset as program on Database
23   end for
24 until Derivation  $\subseteq$  Database

```

(for *main* cf. Algorithm 1) is that the efficiency of Algorithm 1 hinges upon the fact 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.

Therefore, in order to materialize S_Q for each $Q \in L$, Algorithm 2 starts *main*(Q) on the Datalog program P' , which is P where predicates in body atoms have already been replaced by the auxiliary predicates S_Q whenever possible. The auxiliary relations named S_Q with $Q \in L$ are thus empty at first and are gradually filled until $S_Q^{\mathcal{J}_0} = S_Q^{P(\mathcal{J})}$, where \mathcal{J}_0 is \mathcal{J} after Algorithm 2 has terminated. We will, after discussing the algorithm, prove that Algorithm 2 is correct in the sense that $S_Q^{\mathcal{J}_0} = S_Q^{P(\mathcal{J})}$ for all $Q \in L$ after Algorithm 2 has terminated.

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 $R_i(\bar{t}_i)$ with the atom $S_Q(\bar{t}_i)$ if $R_i(\bar{t}_i) \sqsubseteq Q$ (cf. page 3), i.e. if the “answers” to $R_i(\bar{t}_i)$ are also yielded by Q .

Clearly, the result of the rewriting need not to be deterministic in case there are two or more atomic queries $Q_0, Q_1 \in L$ with $R_i(\bar{t}_i) \sqsubseteq Q_0, Q_1$. However, we shall

show that either rewriting is good enough. 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), \mathfrak{J} is extended in an external step (lines 17–19) with all answers for $Q \in L$, which are copied into the auxiliary relation $S_Q^{\mathfrak{J}}$. Since this is repeated between each derivation until no new answers for any $Q \in L$ are yielded, this is equivalent to adding $S_Q(\bar{t}) \leftarrow R(\bar{t})$ for each $Q \in L$ with $Q = R(\bar{t})$ to P' directly.⁴ One can derive from this argument that Algorithm 2 terminates and is sound in the sense that $S_Q^{\mathfrak{J}_0} \subseteq Q^{P(\mathfrak{J})}$, where \mathfrak{J}_0 is the database \mathfrak{J} after Algorithm 2 has terminated. As we shall show in Proposition 2, Algorithm 2 is complete in the sense that, after termination of this algorithm, $S_Q^{\mathfrak{J}_0}$ contains all answers for Q in $P(\mathfrak{J})$.

Example 7. Take the altered program from Example 4 and add the appropriate $S_Q(x, \text{SPO}, y) \leftarrow Q$ with $Q = T(x, \text{SPO}, y)$ to it. In this case we obtain

$$\begin{aligned} T(a, p1, b) &\leftarrow S_Q(p, \text{SPO}, p1) \wedge T(a, p, b) \\ T(x, \text{SPO}, y) &\leftarrow S_Q(x, \text{SPO}, w) \wedge S_Q(w, \text{SPO}, y) \\ S_Q(x, \text{SPO}, y) &\leftarrow T(x, \text{SPO}, y) \end{aligned}$$

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

$$\begin{aligned} T(a, p1, b) &\leftarrow T(p, \text{SPO}, p1) \wedge T(a, p, b) \\ T(x, \text{SPO}, y) &\leftarrow T(x, \text{SPO}, w) \wedge T(w, \text{SPO}, y) \end{aligned}$$

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

Proof. Let P' be the rewritten program P , defined as *NewRuleset* in the pseudocode of Algorithm 2. We have $Q^{P(\mathfrak{J})} \subseteq Q^{P(\mathfrak{J}_0)}$ (monotonicity) and $Q^{P'(\mathfrak{J}_0)} \subseteq S_Q^{\mathfrak{J}_0}$ (line 18). In order to show $Q^{P(\mathfrak{J})} \subseteq S_Q^{\mathfrak{J}_0}$ we show $Q^{P(\mathfrak{J}_0)} \subseteq Q^{P'(\mathfrak{J}_0)}$.

As strategy, we take the Datalog proof tree of an answer to a query $Q \in L$ in $P(\mathfrak{J}_0)$ and show that we can render this proof tree into a Datalog proof tree in $P'(\mathfrak{J}_0)$.

⁴By definition, this would render $S_Q(\bar{t})$ into an idb, which we thus only propose for the sake of explaining correctness.

Assume no new element could be derived from \mathfrak{J}_0 using the program P' but for some $Q \in L$, $\text{main}(Q)$ could derive another yet unknown ground atom from \mathfrak{J}_0 using the original program P . Let hence $\Delta := \bigcup_{Q \in L} Q^{P(\mathfrak{J}_0)} \setminus Q^{P'(\mathfrak{J}_0)}$ and let $R(\bar{a}) \in \Delta$ such that it has a Datalog proof-tree (G, ℓ) in $P(\mathfrak{J}_0)$ of height $m < \omega$, where m is for all atoms from Δ the least height of their Datalog proof-trees in $P(\mathfrak{J}_0)$.

We change the tree (G, ℓ) recursively as follows: If the root v is a leaf, the tree stays unaltered. Otherwise, there is a rule $r \in P$ and an assignment β such that there is a bijection ι between the successor set $E(v)$ and the set of body atoms of r such that $\ell(v') = \beta \circ \iota(v')$ for all $v' \in E(v)$. As we only exchanged predicate names in the rewriting of $r \in P$ to $r' \in P'$, there is a bijection ι' between $E(v)$ and the body atoms of r' and we set $\ell'(v') := \beta \circ \iota'(v')$. For all $v' \in E(v)$ we have $\ell(v') = \ell'(v')$ if the body atom was not replaced during rewriting. If for any $v' \in E(v)$ $\ell'(v')$ has predicate name S_Q for some $Q \in L$, prune its subtree but keep v' . Otherwise recursively continue to change the subtree $\langle v' \rangle$. We thus obtain a new tree G' with a labelling function ℓ' .

Line 18 guarantees that $S_Q^{\mathfrak{J}_0} = Q^{\mathfrak{J}_0}$ for all $Q \in L$ and since (G, ℓ) was chosen minimal, every leaf of (G', ℓ') is labelled with an atom in \mathfrak{J}_0 . Hence (G', ℓ') is a Datalog proof-tree in $P'(\mathfrak{J}_0)$ for $R(\bar{a})$ and so $R(\bar{a}) \notin \bigcup_{Q \in L} Q^{P(\mathfrak{J}_0)} \setminus Q^{P'(\mathfrak{J}_0)}$. A contradiction! \square

5.2. Reasoning with pre-materialized predicates

We show now that replacing body atoms with auxiliary predicates that contain the full materialization of the body atom w.r.t. a given database (i.e. after Algorithm 2), yields the same full materialization of the database as under the original program. We will formally define what conditions must be fulfilled and prove that if they hold, then the two programs will produce the same derivation (Proposition 3). Finally, we point out that the output of Algorithm 2 satisfies these conditions and therefore guarantees the correctness of our entire hybrid approach.

We start by taking an arbitrary Datalog program P and a database \mathfrak{J} . We assume that \mathfrak{J} has already been enriched with the results of the pre-materialization and that S is the name of one of these pre-materialized relations where S is an edb for P .

As an example, assume that this binary relation $S^{\mathfrak{J}}$ contains all tuples (x, y) of the query

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

under the program P .

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

1 Since S is an edb, it does not appear in the head
2 of any rule of P and thus cannot be unfolded. So the
3 evaluation of S during the backward chaining process
4 is reduced to a mere look-up in the database.

5 Replacing an atom in a rule body with an atom contain-
6 ing S , $S(\bar{t})$ say, is harmless only if $S(\bar{t})$ yields the
7 “right” answers. Thus, the question arises which ab-
8 stract conditions must $S(\bar{t})$ satisfy to allow such a re-
9 placement: The answer is that we want a rule to fire
10 under “almost the same” variable assignment as its re-
11 placement, i.e. the rule with the replaced body atom.
12 We will formalize this in the following two paragraphs.

13 Let $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ be a rule in
14 P . We define two queries, one being the body of the
15 rule and one being the body of the rule where for some
16 $i \in \{1, \dots, n\}$ the body atom $R_i(\bar{t}_i)$ is replaced by
17 $S(\bar{t})$ where \bar{t} is some arbitrary tuple having the arity of
18 S . Let $\bar{z} := \bar{t}_0 \cdot \bar{t}_1 \dots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \dots \bar{t}_n$, i.e. the concate-
19 nation of all tuples except \bar{t}_i and
20

$$\begin{aligned} 21 \quad q_0(\bar{z}) &\leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_i(\bar{t}_i) \wedge \dots \wedge R_n(\bar{t}_n) \\ 22 \quad q_1(\bar{z}) &\leftarrow R_1(\bar{t}_1) \wedge \dots \wedge S(\bar{t}) \wedge \dots \wedge R_n(\bar{t}_n) \quad (\diamond) \end{aligned}$$

24 Now, the rule and its replacement fire under *al-*
25 *most the same variable assignment* iff $q_0(\bar{z})^{P(\mathcal{J})} =$
26 $q_1(\bar{z})^{P(\mathcal{J})}$, i.e. q_0 and q_1 yield the same answers under
27 P in \mathcal{J} . We see, that it is “almost the same” variable
28 assignment, as we do not require variable assignments to
29 coincide on \bar{t}_i and \bar{t} . In this way we do not require,
30 e.g., $S^{\mathcal{J}} = R_i^{P(\mathcal{J})}$. S is merely required to contain the
31 *necessary* information. This is important, if we want
32 to apply the substitution to RDF triples, where we lack
33 distinguished predicate names:
34

35 **Example 8.** Since there is only one generic predicate
36 symbol T in RDF, requiring $S^{\mathcal{J}} = T^{P(\mathcal{J})}$ would mean
37 that S contains the complete materialization of \mathcal{J} under
38 P which would render our approach obsolete.

39 Also note that it is not sufficient to merely require
40 $q_0(\bar{t}_0)^{P(\mathcal{J})} = q_1(\bar{t}_0)^{P(\mathcal{J})}$, i.e. that both queries yield
41 the same answer tuples \bar{t}_0 under $P(\mathcal{J})$, as the following
42 example shows.
43

44 **Example 9.** Let the program P which computes the
45 transitive closure of R_0 in R_1 consist of the two rules:
46

$$\begin{aligned} 47 \quad R_1(x, z) &\leftarrow R_1(x, y) \wedge R_0(y, z) \\ 48 \quad R_1(x, y) &\leftarrow R_0(x, y) \end{aligned}$$

50 Consider a database \mathcal{J} with $R_0^{\mathcal{J}} := \{(a, b), (b, c), (b, b),$
51 $(c, c)\}$. In the materialization $P(\mathcal{J})$ of P we expect

52 $R_1^{P(\mathcal{J})} = \{(a, b), (b, c), (a, c), (b, b), (c, c)\}$. Let S
53 have the interpretation $S^{\mathcal{J}} = \{(b, b), (c, c)\}$. Since
54 $R_1^{P(\mathcal{J})}$ is the transitive closure, the following two
55 queries deliver the same answer tuples under $P(\mathcal{J})$, i.e.
56

$$\begin{aligned} 57 \quad q_0(x, z) &\leftarrow R_1(x, y) \wedge R_0(y, z) \\ 58 \quad q_1(x, z) &\leftarrow R_1(x, y) \wedge S(y, z) \end{aligned}$$

60 Yet the program P'

$$\begin{aligned} 61 \quad R_1(x, z) &\leftarrow R_1(x, y) \wedge S(y, z) \\ 62 \quad R_1(x, y) &\leftarrow R_0(x, y) \end{aligned}$$

63 will *not* compute the transitive closure of R_0 in R_1 , as
64 $R_1^{P'(\mathcal{J})} = \{(a, b), (b, c), (b, b), (c, c)\}$.

65 In Proposition 3, we show that we have chosen the
66 correct criterion when requiring that rules must fire
67 under almost the same variable assignments to be re-
68 placements of each other: We show that substituting
69 a body atom $R_i(\bar{t}_i)$ by $S(\bar{t})$, under the condition that
70 the queries in (\diamond) yield the same answer tuples under
71 $P(\mathcal{J})$, generates the same materialization.
72

73 **Proposition 3.** Let P' be the program P where the
74 rule

75 $R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_i(\bar{t}_i) \wedge \dots \wedge R_n(\bar{t}_n) \in P$
76 *has, for some tuple \bar{t} and edb S , been replaced by*
77 $R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge S(\bar{t}) \wedge \dots \wedge R_n(\bar{t}_n)$.

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

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

80 *Proof.* In order to show the implication we assume
81 $q_0(\bar{z})^{P(\mathcal{J})} = q_1(\bar{z})^{P(\mathcal{J})}$. Let T_P and $T_{P'}$ be the im-
82 mediate consequence operators (mentioned on page 4)
83 for each program. We show for all $k < \omega$ and every
84 ground atom $R(\bar{a})$ that if $R(\bar{a}) \in T_P^k(\mathcal{J})$, then there
85 is an $\ell < \omega$ such that $R(\bar{a}) \in T_{P'}^\ell(\mathcal{J})$ and vice versa.
86 Since we start out from the same database \mathcal{J} we have
87 $T_P^0(\mathcal{J}) = T_{P'}^0(\mathcal{J})$ which settles the base case.
88

89 For the step case, let $R(\bar{a}) \in T_{P'}^{k+1}(\mathcal{J})$. Then either
90 $R(\bar{a}) \in T_P^k(\mathcal{J})$ and we are done or there is some rule
91 $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ and some variable
92 assignment β such that $\beta(\bar{t}_0) = \bar{a}$ and $R_j(\beta(\bar{t}_j)) \in$
93 $T_P^k(\mathcal{J})$ for all $j \in \{1, \dots, n\}$.

94 If $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n) \in P$, i.e. none
95 of its body atoms were substituted, the induction hy-
96 pothesis shows for each $j \in \{1, \dots, n\}$ that we can
97 find $\ell_j < \omega$ such that $R_j(\beta(\bar{t}_j)) \in T_{P'}^{\ell_j}(\mathcal{J})$. Let $\ell_0 :=$
98 $\max(\{0\} \cup \{\ell_j \mid 1 \leq j \leq n\})$. Note that we add $\{0\}$
99 for the case where the rule body is empty. In any case,
100 we have $R_j(\beta(\bar{t}_j)) \in T_{P'}^{\ell_0}(\mathcal{J})$ for all $j \in \{1, \dots, n\}$.
101
102

1 Since all premises of this rule are satisfied, there is
2 some $\ell := \ell_0 + 1$ such that $R_0(\beta(\bar{t}_0)) \in T_P^\ell(\mathcal{J})$.

3 If $R(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n) \notin P$ it is a
4 rule where $R_i(\bar{t}_i)$ has been substituted with $S(\bar{t})$. For
5 the assignment β we now know $\beta(\bar{t}_0 \cdot \bar{t}_1 \dots \bar{t}_{i-1} \cdot$
6 $\bar{t}_{i+1} \dots \bar{t}_n) \in q_0(\bar{z})^{P(\mathcal{J})}$. Since $q_0(\bar{z})^{P(\mathcal{J})} = q_1(\bar{z})^{P(\mathcal{J})}$
7 we know that there is some assignment β' , which co-
8 incides with β on $(\bar{t}_0 \cdot \bar{t}_1 \dots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \dots \bar{t}_n)$ such that
9 $\beta'(\bar{t}) \in S^{P(\mathcal{J})}$.

10 Hence $R_j(\beta'(\bar{t}_j)) \in T_P^k(\mathcal{J})$ for all $j \in \{1, \dots, n\} \setminus$
11 $\{i\}$ and $S(\beta'(\bar{t})) \in T_P^0(\mathcal{J})$ since S is an edb predicate.
12 The induction hypothesis yields some $\ell_j < \omega$ for each
13 $j \in \{1, \dots, n\} \setminus \{i\}$ such that $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_j}(\mathcal{J})$.
14 Let $\ell_0 := \max(\{0\} \cup \{\ell_j \mid 1 \leq j \leq n \text{ and } j \neq i\})$,
15 then $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_0}(\mathcal{J})$ for all $j \in \{1, \dots, n\} \setminus \{i\}$
16 and $S(\beta'(\bar{t})) \in T_P^0(\mathcal{J})$. Since all premises of this
17 rule are satisfied, there is some $\ell := \ell_0 + 1$ such
18 that $R_0(\beta'(\bar{t}_0)) \in T_P^\ell(\mathcal{J})$. As β coincides with β'
19 also on \bar{t}_0 , i.e. $\beta'(\bar{t}_0) = \bar{a}$, we have in particular
20 $R(\bar{a}) \in T_P^\ell(\mathcal{J})$.

21 This shows $R^{P(\mathcal{J})} \subseteq R^{P'(\mathcal{J})}$ for all predicate names
22 $R \in \text{PRED}$. For the converse, we merely show the
23 case of the substituted rule: Assume $R(\bar{a}) \in T_P^{k+1}(\mathcal{J})$
24 and there is an assignment β' such that $\beta'(\bar{t}_0) = \bar{a}$ and
25 $R_j(\beta'(\bar{t}_j)) \in T_P^k(\mathcal{J})$ for all $j \in \{1, \dots, n\} \setminus \{i\}$ as
26 well as $\beta'(\bar{t}) \in S^{P'(\mathcal{J})}$.

27 The induction hypothesis yields for each $j \in$
28 $\{1, \dots, n\} \setminus \{i\}$ some $\ell_j < \omega$ with $R_j(\beta'(\bar{t}_j)) \in$
29 $T_P^{\ell_j}(\mathcal{J})$. Since S is an edb predicate for P , we have
30 $S(\beta'(\bar{t})) \in T_P^0(\mathcal{J})$. Hence for $\ell_0 := \max(\{0\} \cup \{\ell_j \mid$
31 $1 \leq j \leq n \text{ and } j \neq i\})$ we have $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_0}(\mathcal{J})$
32 for all $j \in \{1, \dots, n\} \setminus \{i\}$ and $S(\beta'(\bar{t})) \in T_P^0(\mathcal{J})$.

33 This implies $\beta'(\bar{t}_0 \cdot \bar{t}_1 \dots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \dots \bar{t}_n) \in$
34 $q_1(\bar{z})^{P(\mathcal{J})}$ and since $q_0(\bar{z})^{P(\mathcal{J})} = q_1(\bar{z})^{P(\mathcal{J})}$ there is an
35 assignment β coinciding on $(\bar{t}_0 \cdot \bar{t}_1 \dots \bar{t}_{i-1} \cdot \bar{t}_{i+1} \dots \bar{t}_n)$
36 with β' such that $R_i(\beta(\bar{t}_i)) \in T_P^{j_0}(\mathcal{J})$ for some $j_0 <$
37 ω . Let $\ell_1 := \max\{\ell_0, j_0\}$ then $R_j(\beta'(\bar{t}_j)) \in T_P^{\ell_1}(\mathcal{J})$
38 for all $j \in \{1, \dots, n\}$. Since all premises of the rule
39 $R_0(\bar{t}_0) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_n(\bar{t}_n)$ are satisfied, there
40 is some $\ell := \ell_1 + 1$ such that $R_0(\beta(\bar{t}_0)) \in T_P^\ell(\mathcal{J})$,
41 which shows, as β coincides on \bar{t}_0 with β' that $R(\bar{a}) \in$
42 $T_P^\ell(\mathcal{J})$.

43 Together with $R^{P(\mathcal{J})} \subseteq R^{P'(\mathcal{J})}$ this shows $R^{P(\mathcal{J})} =$
44 $R^{P'(\mathcal{J})}$ for all predicate names $R \in \text{PRED}$ and hence
45 that $P(\mathcal{J}) = P'(\mathcal{J})$. \square

46
47
48 It now becomes clear, how Algorithm 2 and Propo-
49 sition 3 fit together: For a given database \mathcal{J} and a set
50 of atomic queries L , Algorithm 2 computes for each
51 $Q \in L$ the query answers under the program P , which

are stored in the relation $S_Q^{\mathcal{J}_0}$, where \mathcal{J}_0 is \mathcal{J} after Al-
gorithm 2 has finished. These S_Q are edbs for P .

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

$$q_0(\bar{z}) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge R_i(\bar{t}_i) \wedge \dots \wedge R_n(\bar{t}_n)$$

$$q_1(\bar{z}) \leftarrow R_1(\bar{t}_1) \wedge \dots \wedge S_Q(\bar{t}_i) \wedge \dots \wedge R_n(\bar{t}_n)$$

Proposition 3 guarantees that the substitution of $R_i(\bar{t}_i)$
by $S_Q(\bar{t}_i)$ in rule r is harmless w.r.t. \mathcal{J} . By applying
this argument iteratively, one eventually obtains a pro-
gram P' in which all pre-computed atoms have been
replaced and which yields the same materialization for
 \mathcal{J}_0 as P .

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

6. Hybrid reasoning for OWL RL

In the previous sections, we described the two
main components of our method which consist of the
backward-chaining algorithm used to retrieve infer-
ences as well as the pre-materialization procedure to-
gether with the predicate replacement.

We now discuss the implementation of the OWL RL
rules using our approach. In fact, while our method can
in principle be applied to any Datalog program, our
implementation heavily relies on the fact that our pro-
gram consists of inference rules on RDF data and thus,
our prototype is unable to execute generic Datalog pro-
grams.

The official OWL RL rule set contains 78 rules, for
which the reader is referred to the official document
overview [13]. With selected examples from [13] we
illustrate some key features of our algorithm.

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

- All the rules whose purpose is to derive an incon-
sistency, i.e. rules with predicate *false* in the head
of the rule. We do not consider them because our
objective is to derive new triples rather than iden-
tify an inconsistency;

Table 2

Triple patterns that are pre-materialized considering the OWL RL rules (1 is an abbreviation for the typed literal "1"^^xsd:nonNegativeInteger)

OWL RL	pD^*	RDFS
		(?X rdfs:subPropertyOf ?Y) (?X rdfs:subClassOf ?Y) (?X rdfs:domain ?Y) (?X rdfs:range ?Y)
		(?P rdf:type owl:FunctionalProperty) (?X owl:allValuesFrom ?Y) (?P rdf:type owl:InverseFunctionalProperty) (?X owl:inverseOf ?Y) (?P rdf:type owl:TransitiveProperty) (?X rdf:type owl:Class) (?P rdf:type owl:SymmetricProperty) (?X rdf:type owl:Property) (?X owl:equivalentClass ?Y) (?X owl:onProperty ?Y) (?X owl:hasValue ?Y) (?X owl:equivalentProperty ?Y) (?X owl:someValuesFrom ?Y)
		(?X owl:propertyChainAxiom ?Y) (?X owl:hasKey ?Y) (?X owl:intersectionOf ?Y) (?X owl:unionOf ?Y) (?X owl:oneOf ?Y) (?X owl:maxCardinality 1) (?X owl:maxQualifiedCardinality 1) (?X owl:onClass ?Y) (?X rdf:type owl:Class) (?X rdf:type owl:DatatypeProperty) (?X rdf:type owl:ObjectProperty)

- All the rules which have an empty body. The rules cannot be triggered during the unfolding process of backward-chaining. These rules include those that encode the semantics of datatypes and therefore our implementation does not support datatypes;
- The rules that exploit the *owl:sameAs* transitivity and symmetry.⁵ 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.⁶

This excludes 30 out of 78 rules, leaving 48 rules.

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. Table 2 reports the set of the patterns that are pre-materialized using our method described in Section 5.

⁵These are all rules reported in Table 4 of [13].

⁶This procedure is explained in detail in [20].

Singling out exactly those triple patterns from Table 2 is motivated by the fact that these patterns appear in many OWL rules, and is grounded on the assumptions that (i) their answer sets are very small compared to the entire input, and (ii) they 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 among their body atoms are substituted replacing the pre-materialized pattern with its corresponding auxiliary relation as justified by Proposition 3.

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 10. Consider (scm-sco) from Table 9 in [13]:

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

can be replaced according to Proposition 3 by r' :

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

where the answers to $T(x, SCO, y)$ under \mathcal{J} are contained in $S_{sco}^{\mathcal{J}}$. By adding the rule

$$r'': T(x, SCO, z) \leftarrow S_{sco}(x, SCO, z)$$

every rule whose head atom unifies with $T(x, SCO, y)$ can be deleted. Adding r'' is harmless; we can render r' into r'' using Proposition 3: Since S_{sco} is transitive we may replace an imaginary conjunction with \top (the atom which is always true) in r' with $S_{sco}(x, SCO, z)$. By two more applications of Proposition 3, we can successively replace $S_{sco}(x, SCO, y)$ and $S_{sco}(y, SCO, z)$ each by \top obtaining r'' .

This allows us to remove from unfolding 20 of the 48 rules after the pre-materialization is completed.

On the implementation of RDF lists Some of the inference rules in the OWL RL rule set use RDF lists to allow for an arbitrary number of antecedents. The RDF lists cannot be represented in Datalog in a straightforward way since they rely on *rdf:first* and *rdf:rest* triples to represent the elements of the list. Therefore, they need to be processed differently.

In our implementation, at every step of the pre-materialization procedure, we launch two additional queries to retrieve all the (inferred and explicit) *rdf:first* and *rdf:rest* triples with the purpose of constructing such lists. Once we have collected them, we perform a join with the other schema triples, and determine the sequence of elements by repeatedly joining the *rdf:first* and *rdf:rest* triples. After this operation is completed,

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

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.

6.1. 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 empirically analyzing some example queries, we detected two major duplicate sources in our experiments and devised strategies to avoid them. The first comes from the nature of the rule set. The second comes from the input data.

First source of duplicates 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 [13]:

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

We analyze each of these three cases below.

Let $S_{eqp}^{\mathcal{J}}$ be the pre-materialization of the atomic query $T(x, EQP, y)$. Together, the rules *scm-eqp1* and *scm-eqp2* render $S_{eqp}^{\mathcal{J}}$ symmetric. Hence the query

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

yields the same results under $P(\mathcal{J})$ as

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

Proposition 3 allows $S_{eqp}(p_1, EQP, p_2)$ in the rewritten rule *prp-eqp1'*:

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

to be replaced by $S_{eqp}(p_2, EQP, p_1)$, which is, up to variable renaming, the rule *prp-eqp2'*:

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

and thus effectively deleting *prp-eqp1'* from the rule set.

Similarly, rules *scm-eqc1* and *scm-eqc2* render the results of the pre-materialized query $T(x, EQC, y)$ symmetric allowing to delete *cax-eqc1* or rather its rewriting *cax-eqc1'* in a similar fashion.

Due to the lack of an appropriate rule, the pre-materialized query $T(x, INV, y)$ need not yield a symmetric result, although the intended relation, “being the inverse of each other”, is symmetric. We can first observe that Proposition 3 allows us to replace the rules *prp-inv1* and *prp-inv2* by their rewritings

$$\begin{aligned} T(y, p, x) &\leftarrow T(x, q, y) \wedge S_{inv}(p, INV, q) \\ T(y, p, x) &\leftarrow T(x, q, y) \wedge S_{inv}(q, INV, p) \end{aligned}$$

which defuses the idb atom $T(q, INV, p)$ into the harmless edb S_{inv} , for which $S_{inv}^{\mathcal{J}}$ 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

$$T(x, p, y) \leftarrow T(x, q, y) \wedge S'_{inv}(p, INV, q)$$

with S'_{inv} being the symmetric closure of $S_{inv}^{\mathcal{J}}$. It is now not difficult to see, that every model of P' is a model of P'' and vice versa. In particular the full materialization $P(\mathcal{J})$ is equal to the full materialization $P''(\mathcal{J})$. 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 source of duplicates The second type of duplicate generation 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, which is a realistic data set that we used to evaluate our approach. In this data set there is the triple $T(SCO, TYPE, TRANS)$ which states that the *rdfs:subClassOf* predicate is transitive.

In this case, during the pre-computation phase the query $T(x, SCO, y)$ will be launched several times, and each time the reasoner will trigger the application of both the rules *scm-sco* and *prp-trp*.

However, since the application of these two rules will lead to the same derivation, the computation is redundant and inefficient. To detect such cases, we apply a special algorithm when the system is starting up and initializing the rule set. A complete description of this algorithm is outside the scope of this paper and we will simply illustrate the main idea behind it.

Basically, this algorithm compares each rule with every other rule in order to identify under which con-

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

ditions the two will produce the same output to a given query. For example, the rules *scm-sco* and *prp-trp* will produce the same derivation if (i) the input contains the triple $T(SCO, TYPE, TRANS)$ and if (ii) there is a query with *SCO* as a predicate.

In order to verify that this is the case, the algorithm checks whether the triple $T(SCO, TYPE, TRANS)$ exists in the input and whether 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 a matching exists, then the two rules are equivalent. In our example, the algorithm will find out that the rule *prp-trp* is equivalent to *scm-sco* if we replace *?p* with *SCO*. Therefore, if there is an input query with *SCO* as predicate, the system will execute only one of the two rules, avoiding in this way a duplicated derivation.

7. Evaluation

We have implemented our approach in a Java prototype called *QueryPIE*⁷ and we have evaluated the performance using one machine of the DAS-4 cluster,⁸ 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, configured in RAID-0 mode.

To the best of our knowledge, there is not (yet) a proper benchmark for reasoning over large data sets that extensively uses all the new features introduced with the OWL RL language. Therefore, we chose to evaluate our method using the most common and large-scale data sets currently available in order to evaluate how hybrid reasoning would perform on *current* data and *realistic* queries. Because of this, we use two data sets as input: LUBM [9], which is one of the most popular benchmarks for OWL reasoning and LLD (Linked Life Data),⁹ which is a curated collection of real-world data sets in the bioinformatics domain.

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

⁷The experimental code is available at: <https://github.com/jrbrn/querypie>.

⁸<http://www.cs.vu.nl/das4>

⁹<http://www.linkedlifedata.com/>

Table 3

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

Dataset	Reasoning time		N. iterations	N. derived triples
	Our approach	Full materialization		
LUBM	1.0 s	4d4h16m	4	390
LLD	16 m	5d10h45m	7	10 millions

sets were compressed using the procedure described in [21].

The QueryPIE prototype uses six indices stored alongside with the triple permutations on disk using an optimized B-Tree data structure. During the reasoning process, the inferred triples are stored and cached in the main memory. Furthermore, the content of the pre-materialization is indexed at every iteration to facilitate the retrieval of the *lookup* function.

The rest of this section is organized as follows. First, in Section 7.1 we report on a set of experiments to evaluate the performance of the pre-materialization phase. Next, in Section 7.2 we focus on the performance of the backward-chaining approach and analyze its performance on some example queries.

7.1. Performance of the pre-materialization algorithm

We launch the pre-materialization algorithm on the two data sets 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 also include 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 prematerialization is about three orders of magnitude faster for the LUBM data set than for LLD. The cause for 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 derives 390 triples, needing four iterations to reach a fix point. In the other case, the pre-materialization required 7 iterations and returned about 10 million triples.

We compare the cost of performing the partial closure against the cost of a full materialization, which is currently considered as state of the art in the field of large scale OWL reasoning. To this end, the closest ap-

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

Table 4
List of example queries

ID	Dataset	Query
Pattern 1	LUBM	?x ?y <http://www.Department0.University0.edu/GraduateCourse0>
Pattern 2	LUBM	?x <lubm:subOrganizationOf> <http://www.University0.edu>
Pattern 3	LUBM	<http://.../GraduateStudent124> <lubm:degreeFrom> <http://www.University114.edu>
Pattern 4	LUBM	?x ?y <http://www.Department0.University0.edu/AssistantProfessor0>
Pattern 5	LUBM	?x <lubm:memberOf> <http://www.Department0.University0.edu>
Pattern 6	LUBM	?x <rdf:type> <lubm:Department>
Pattern 7	LLD	?x ?y <lifeskim:mentions>
Pattern 8	LLD	?x <lifeskim:mentions> <http://linkedlifedata.com/resource/umls/id/C0439994>
Pattern 9	LLD	<http://.../resource/pubmed/id/15964627> ?x ?y
Pattern 10	LLD	?x ?y <http://purl.uniprot.org/go/0006952>
Pattern 11	LLD	?x ?y <http://linkedlifedata.com/resource/umls/id/C0439994>
Pattern 12	LLD	?x <http://www.biopax.org/release/biopax-level2.owl#NAME> ?y

proach we can use for a comparison is WebPIE [20], 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 (the estimation is in line with the performance of WebPIE which has shown linear scalability in [20]).

The runtime of the complete materialization performed with this method is reported in the third column of Table 3. Note 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. Note, however, that there could be cases where our approach becomes particularly inefficient if our backward-chaining algorithm needs to re-derive the same intermediate triples to answer different schema patterns.

In any case, the advantage of performing only a pre-materialization comes at a price: while after a complete materialization reasoning is no longer needed, our backward-chaining algorithm still has to perform some inference at query time. The impact of this operation on the query-time performance is analyzed in the next section.

7.2. Performance of the reasoning at query time

In order to analyze the performance of reasoning at query time, we launch some example queries after computing the closure using our backward-chaining

algorithm to retrieve the results. For this purpose, we select six example queries for both the LUBM and LLD data sets reported in Table 4.

While LUBM provides an official set of queries for benchmarking, there is unfortunately no official set of queries that can be used for benchmarking the performance on the LLD data set. Therefore, we took some queries that are reported on the official web page of the LLD data set and modified them so that they 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 perform 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.

7.2.1. Absolute response time

We report in Table 5 the execution times obtained launching the selected example queries in Table 4. In

Table 5

Runtime of the queries in Table 4 on the LUBM and LLD data sets

Q.	Runtime (ms)		Derived triples		I/O access	
	Cold	Warm	Total	Output	#	MB
1	60.43	6.39	5	5	43	8
2	1099.28	129.31	463	239	12	205
3	49.18	6	3	1	18	5
4	73.06	11.17	37	29	86	8
5	118.71	13.97	1480	719	18	8
6	4026.27	2590.27	1599987	1599987	2	12
7	228.26	214.57	0	0	670	23
8	23.74	6.29	4466	4466	1	4
9	7064.04	609.4	140	128	3540	105
10	2535.38	1103.48	28446	26860	14372	337
11	2613.37	1883.14	8546	4504	15128	64
12	2334.70	2059.20	1187944	1187944	1	10

the second and third columns we report both the cold and warm runtime.¹⁰ 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, we also measure with the cold runtime 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. During this execution the data is already cached in memory and the Java VM has already initialized the internal data structures, so 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 for estimating the impact of reasoning at query time. While querying without reasoning only requires a data lookup, the backward-chaining algorithm might require to access the database multiple times. For example, in order to answer query 11 the program had to access the data indices about 15000 times.

Using the results reported in Table 5, we make several observations: First, we notice that the cold runtime is in general significantly higher than the warm

¹⁰Please note that the reported runtime does not include the time required to compress/decompress the numerical terms to their string counterpart.

runtime between one and two orders of magnitude. This is primarily due to the time consuming I/O access to disk especially because reasoning requires to read in different locations of the data indices. 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 presents query 10 where the program reads from disk about 337 MB of data. We conclude from these results that the performance of the program is essentially I/O bounded, if the data is stored on disk. After the data is loaded in memory, the execution time drops in half of the experiments by a factor between 6.5 and 11.6 and in the other half of the experiments by a factor between 1.1 and 3.8.

Another factor that affects the performance is the number of inferred triples that are calculated during the execution of the query. In fact, we notice that the absolute performance is lower in case a large number of triples is either inferred or retrieved from the database. This 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 a 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 can be interactively used since few seconds are acceptable in most scenarios.

In Section 7.2.3 we compare the response times to those without reasoning in order to have a better overview of the overall performance and understand what the overhead induced by reasoning at query time is.

7.2.2. Reduction of the proof tree

Our approach relies on the pre-materialization of some selected queries for a variety of purposes such as performing efficient sideways information passing, ex-

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

Table 6

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

Q.	# Leaves proof tree		Reduction ratio
	W/o pre-comp.	Our approach	
1	16	4	4.0
2	2	1	2.0
3	12	3	4.0
5	26	7	3.7

cluding rules that derive duplicates, and to reduce the size of the proof tree during query-time.

In this section, we evaluate the effective reduction of proof-tree size obtained by avoiding performing inference on the pre-materialized patterns. However, we cannot completely ignore the pre-materialization since some optimizations (e.g. the pruning strategy) are necessary to avoid an explosion of the proof-tree. To overcome this problem, we have manually analyzed the execution of the LUBM queries with our prototype on a much smaller data set (of about 100 thousand triples) and manually constructed the proof trees simulating the case without pre-materialization (note that we excluded queries 4 and 6 since in these cases reasoning did not contribute to derive new answers).

In principle, we identify for each query those rules which produce its answers and for each pre-materialized body atom, we add the corresponding branch that was generated when that query was calculated during the pre-materialization phase.

We report the results of this analysis in Table 6. 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 delivered by this method of evaluation must be seen as an underestimate, because we did 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.

7.2.3. 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 launch a number of experiments activating different types of reasoning at query time and report the warm runtime in Table 7.

We proceed 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 reissue the queries activating only the RDFS rules (in the third column), then the pD^* rules and finally the OWL RL ones.

The results reported under the “Ins.” columns (“Ins.” means insertion) are 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 repeat 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.

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. This behavior is 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.

Furthermore, we notice in our experiments 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 6.1, we tackled this problem by proposing some initial algorithms to limit the number of duplicates. However, our work in this respect is still preliminary 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.

Finally, we can compare the response times reported in the third column with the ones of the penultimate column to compare the performance of the reason-

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

Table 7
Runtime (in ms.) of the example queries using different sets of rules

Q.	Only lookup		RDFS		pD*		OWL RL	
	No Ins.	Ins.	No Ins.	Ins.	No Ins.	Ins.	No Ins.	Ins.
1	0.81	0.83	1.88	1.79	5.4	6.13	6.39	5.89
2	0.82	1.51	1.56	2.83	128.78	131.05	129.31	138.53
3	0.82	0.83	3.55	2.72	5.50	4.51	6	4.83
4	0.88	0.94	2.01	2.32	10.06	9.48	11.17	10.63
5	1.5	1.61	7.01	4.95	13.58	10.52	13.97	10.8
6	405.42	418.38	2605.68	2630.08	2608.20	2619.17	2590.27	2618.66
7	0.77	0.79	176.19	1.26	203.23	17.93	214.57	16.78
8	1.96	1.89	6.23	6.34	6.39	6.46	6.29	6.36
9	0.84	0.90	262.7	46.53	590.34	277.55	609.4	277.02
10	7.90	7.29	212.57	115.16	903.31	814.95	1103.48	1053.33
11	1.85	1.93	200.55	8.35	1695.73	1468	1883.14	1529.64
12	338.14	337.41	2129.49	2044.34	2055.02	2077.55	2059.2	2062.65

ing 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 orders 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 pre-computation phase is no longer needed.

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 a small set of 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.

8. Related work

Applying rules with a top-down method like backward-chaining is a well-known technique in rule-based languages like Datalog [6]. Datalog is a generic and powerful language that can handle an arbitrary number of rules using predicates of any arity. In our work, we optimized the implementation to exploit the char-

acteristics of RDF data and to execute a standard rule set, ignoring features of Datalog that are not necessary to execute the OWL rules. Therefore, and since there is, to the best of our knowledge, no Datalog engine that can load billions of triples, a direct comparison between our work and similar Datalog engines such as IRIS [5] or Jena [12] is not possible.

The backward-chaining algorithm that we present in Section 4 is inspired by the QSQ and the semi-naive evaluation algorithms which are well-known techniques in logic programming. A similar termination condition to ours is employed also in the RQA/FQI algorithm [14].

In our approach, we exploit the availability of the pre-computation using a *sideways information passing (SIP)* technique [3] 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 SIP at compile-time to construct rules which are later used in a bottom-up fashion, 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 [10] the authors propose two adaptive SIP strategies where information is passed adaptively between operators that are executed in parallel.

A similar technique to our method is memoing [26]. Memoing is a technique where queries that are frequently requested are cached to improve the performance. The difference between memoing and our work is that in memoing caching is dynamic, while in our approach caching is static: we manually select which queries that are to be cached.

1 Some RDF Stores support various types of inference. 4store [17] applies the RDFS rules with
 2 backward-chaining. Virtuoso [8] supports the execution of few (but not all) OWL RL rules. BigOWLIM
 3 [4] is an RDF store that supports the OWL 2 RL rule set by performing a full materialization when the data
 4 is being loaded. Another database system that performs OWL RL reasoning in a similar way is Oracle:
 5 In [11] the authors describe their approach reporting the performance of the inference over up to seven bil-
 6 lion triples. An approach in which the OWL RL rules are used is presented in [18] where the authors have
 7 encoded OWL RL reasoning in the context of embedded devices, and therefore optimizing the computation
 8 for devices with limited resources.

9 Some work has been presented to distribute the reasoning process using supercomputers or clusters
 10 of machines. In our previous work we used the MapReduce programming model to improve the scal-
 11 ability [20]. In [27], the authors implement RDFS reasoning using the Opteron blade cluster. To the
 12 best of our knowledge, none of the mentioned approaches supports a similarly large subset of OWL RL
 13 rules.

14 Implicit information can be derived not only with rule-based techniques. In [15], the authors focus on on-
 15 tology based query answering using the OWL 2 QL profile [13] and present a series of techniques based on
 16 query rewriting to improve the performance. While we demonstrate inference over a much larger scale, a di-
 17 rect comparison of our technique with this work is difficult since both the language and reasoning techniques
 18 are substantially different.

19 A series of work has been done on reasoning using the OWL EL profile. This language is targeted to do-
 20 mains in which there are ontologies with a very large number of properties and/or classes. [7] presented an
 21 extensive survey of the performance of OWL EL reasoners analyzing tasks like classification or consis-
 22 tency checking. Again, the different reasoning tasks and considered language makes a direct comparison
 23 difficult for our approach.

24 9. Conclusions

25 Until now, all inference engines that can handle reasonably expressive logics over very large triple stores
 26 (in the orders of billion of triples) have deployed full materialization. In the current paper we have broken

27 with this mold, showing that it is indeed possible to do efficient backward-chaining over large and reasonably
 28 expressive knowledge bases.

29 The key to our approach is to pre-compute a small number of inferences which appear very frequently
 30 in the proof tree. This of course re-introduces some amount of pre-processing, but this computation is mea-
 31 sured in terms of minutes, instead of the hours needed for the full closure computation.

32 By pre-materializing part of the inferences upfront instead of during query-time, we are able to
 33 introduce a number of optimizations which exploit the pre-computation to improve the performance dur-
 34 ing query-time. To this end, we adapted a standard backward-chaining algorithm like QSQ to our use case
 35 exploiting the parallelization of current architectures.

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

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

44 Obvious next steps in future work would be to investigate how our approach can further scale in terms
 45 of data size especially when including those rules from OWL 2 RL which we have not considered so far. We
 46 also need a deeper understanding of which and how properties of the knowledge base influence both the
 47 cost of the limited forward computation and the size of the inference tree. It is worth to explore whether re-
 48 lated techniques such as ad-hoc query-rewriting like the one presented in [15] can be exploited to further
 49 improve the performance.

50 To the best of our knowledge, this is the first time that complex backward-chaining reasoning over real-
 51 istic OWL knowledge bases in the order of ten billion triples has been realized. Our results show that this ap-
 52 proach is feasible, opening the door to reasoning over much more dynamically changing data sets than what
 53 was possible until now.

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102

Acknowledgments

We would like to thank Stefan Schlobach, Barry Bishop, Boris Motik, and Ian Horrocks for providing useful comments and advice. We also would like to thank Cerial Jacobs for the support in developing and debugging the prototype used in the evaluation.

A sincere thanks for the long and detailed reviews goes to the reviewers of this article, Aidan Hogan, Matthias Knorr, Raghava Mutharaju, and Peter Patel-Schneider 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), the COMMIT project and by the SEALS Project.

References

- [1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Vol. 8, Addison-Wesley Reading, 1995.
- [2] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, Magic sets and other strange ways to implement logic programs, in: *Proc. of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS '86*, New York, NY, USA, ACM, 1986, pp. 1–15.
- [3] C. Beeri and R. Ramakrishnan, On the power of magic, *The Journal of Logic Programming* **10**(3) (1991), 255–299.
- [4] B. Bishop and S. Bojanov, Implementing OWL 2 RL and OWL 2 QL rule-sets for OWLIM, in: *OWLED*, M. Dumontier and M. Courtot, eds, CEUR Workshop Proc., Vol. 796, CEUR-WS.org, 2011.
- [5] B. Bishop and F. Fischer, IRIS – Integrated Rule Inference System, in: *AREa2008*, F. van Harmelen, A. Herzig, P. Hitzler, Z. Lin, R. Piskac, and G. Qi, eds, CEUR Workshop Proc., Vol. 350, CEUR-WS.org, 2008.
- [6] S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases*, Springer, 1990.
- [7] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer, Comparison of reasoners for large ontologies in the OWL 2 EL profile, *Semantic Web* **2**(2) (2011), 71–87.
- [8] O. Erling and I. Mikhailov, SPARQL and scalable inference on demand, available at http://www.openlinksw.co.uk/virtuoso/Whitepapers/pdf/Scalable_Inference.pdf.
- [9] Y. Guo, Z. Pan, and J. Heflin, LUBM: A benchmark for OWL knowledge base systems, *Web Semantics: Science, Services and Agents on the World Wide Web* **3**(2) (2005), 158–182.
- [10] Z. Ives and N. Taylor, Sideways information passing for push-style query processing, in: *Data Engineering. IEEE International Conference 24th 2008*, IEEE, 2008, pp. 774–783.
- [11] V. Kolovski, Z. Wu, and G. Eadon, Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system, in: *The Semantic Web – ISWC 2010*, Lecture Notes in Computer Science, Vol. 6496, Springer, Berlin Heidelberg, 2010, pp. 436–452.
- [12] B. McBride, Jena: A semantic web toolkit, *Internet Computing* **6**(6) (2002), 55–59.
- [13] B. Motik, B. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, OWL2 web ontology language: Profiles, W3C Recommendation, 2009.
- [14] W. Nejdl, Recursive strategies for answering recursive queries – The RQA/FQI strategy, in: *Proc. of the 13th International Conference on Very Large Data Bases, VLDB '87*, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 1987, pp. 43–50.
- [15] H. Pérez-Urbina, I. Horrocks, and B. Motik, Efficient query answering for OWL 2, in: *The Semantic Web – ISWC 2009*, Lecture Notes in Computer Science, Vol. 5823, Springer, Berlin Heidelberg, 2009, pp. 489–504.
- [16] E. Prud'hommeaux and A. Seaborne, SPARQL query language for RDF, W3C Recommendation, 2008.
- [17] M. Salvadores, G. Correndo, S. Harris, N. Gibbins, and N. Shadbolt, The design and implementation of minimal RDFS backward reasoning in 4store, in: *The Semantic Web: Research and Applications*, Lecture Notes in Computer Science, Vol. 6644, Springer, Berlin Heidelberg, 2011, pp. 139–153.
- [18] C. Seitz and R. Schönfelder, Rule-based OWL reasoning for specific embedded devices, in: *The Semantic Web – ISWC 2011*, Lecture Notes in Computer Science, Vol. 7032, Springer, Berlin Heidelberg, 2011, pp. 237–252.
- [19] J. Urbani, F. Harmelen, S. Schlobach, and H. Bal, QueryPIE: Backward reasoning for owl horst over very large knowledge bases, in: *The Semantic Web – ISWC 2011*, Lecture Notes in Computer Science, Vol. 7031, Springer, Berlin Heidelberg, 2011, pp. 730–745.
- [20] J. Urbani, S. Kotoulas, J. Maassen, F.V. Harmelen, and H. Bal, WebPIE: A web-scale parallel inference engine using MapReduce, *Web Semantics: Science, Services and Agents on the World Wide Web* **10**(0) (2012), 59–75.
- [21] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal, Scalable RDF data compression with MapReduce, *Concurrency and Computation: Practice and Experience* **25**(1) (2013), 24–39.
- [22] M.H. Van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *Journal of the ACM* **23**(4) (1976), 733–742.
- [23] L. Vieille, Database-complete proof procedures based on SLD resolution, in: *Logic Programming: Proc. of the Fourth International Conference (Vol. 1)*, Cambridge, MA, MIT Press, 1987, pp. 74–103.
- [24] L. Vieille, Recursive axioms in deductive databases: The query/subquery approach, in: *Expert Database Systems, Proc. from the First International Conference (1986)*, Redwood City, CA, USA, Benjamin-Cummings Publishing Co., Inc., 1987, pp. 253–267.
- [25] L. Vielle, Recursive query processing: The power of logic, *Theoretical Computer Science* **69**(1) (1989), 1–53.
- [26] D. Warren, Memoing for logic programs, *Communications of the ACM* **35**(3) (1992), 93–111.
- [27] J. Weaver and J. Hendler, Parallel materialization of the finite RDFS closure for hundreds of millions of triples, in: *The Semantic Web – ISWC 2009*, Lecture Notes in Computer Science, Vol. 5823, Springer, Berlin Heidelberg, 2009, pp. 682–697.