Parallel Tractability of Ontology Materialization: Technique and Practice

Zhangquan Zhou[a,*], Guilin Qi[a], Birte Glimm[b]

[a] *School of Computer Science, Southeast University, China*
[b] *Institute of Artificial Intelligence, University of Ulm*

Abstract

Materialization is an important reasoning service for many ontology-based applications, but the rapid growth of semantic data poses the challenge to efficiently perform materialization on large-scale ontologies. Parallel materialization algorithms work well for some ontologies, although the reasoning problem for the used ontology language is not in NC, i.e., the theoretical complexity class for *parallel tractability*. This motivates us to study the problem of *parallel tractability* of ontology materialization from a theoretical perspective. We focus on the datalog rewritable ontology languages DL-Lite and Description Horn Logic (DHL) and propose algorithms, called NC algorithms, to identify classes of ontologies for which materialization is tractable in parallel. To verify the practical usability of the above results, we analyze different benchmarks and real-world datasets, including LUBM and the YAGO ontology, and show that for many ontologies expressed in DHL materialization is tractable in parallel. The implementation of our optimized parallel algorithm shows performance improvements over the highly optimized state-of-the-art reasoner RDFox on ontologies for which materialization is tractable in parallel.

Keywords: ontology, materialization, datalog, parallel tractability, NC complexity

1. Introduction

The Web Ontology Language (OWL)[1] is an important standard for ontology languages in the Semantic Web and knowledge-based applications. In many of these applications, *materialization* plays an important role, which is the reasoning task of computing all implicit *facts* that follow from a given ontology [1]. Ontology developers and users employ materialization to optimize tasks such as query answering, ontology diagnosis or debugging. Since large amounts of semantic data are being generated at an increasing pace by sensor networks, government authorities and social media [2, 3], it is challenging to conduct materialization on such large-scale ontologies efficiently.

For the ontology language RDFS and its extended fragments, approaches for parallel reasoning and for employing parallel computing platforms exist [4, 5, 6]. Several optimization

*Corresponding author
Email addresses: quanzz1129@gmail.com (Zhangquan Zhou), gqi@seu.edu.cn (Guilin Qi), birte.glimm@uni-ulm.de (Birte Glimm)

[1] The latest version is OWL 2: http://www.w3.org/TR/owl2-overview/

strategies, e.g., dictionary encoding, balancing workload and data partitioning, are further studied to enhance parallel RDFS reasoning. There are also parallel implementations for scalable reasoning over ontologies that use highly expressive ontology languages [7, 8]. The different approaches utilize different kinds of computing platforms to make reasoning tasks more efficient in parallel, e.g., supercomputers [9, 10], MapReduce [11] and GPU servers [12].

The effectiveness of parallel reasoning for the above mentioned techniques is empirically verified on different test datasets. However, materialization is not tractable in parallel for most of the popular ontology languages with PTime-complete or higher complexity [13]. In particular, this is the case for RDFS and datalog rewritable ontology languages and, therefore, the efficiency of reasoning may not be improved using a parallel implementation. A possible reason for the apparent high performance of parallel reasoning is that the utilized test datasets do not fall into the worst cases in terms of computational complexity. Also some well-known, large-scale ontologies such as YAGO show good performance for parallel reasoning [14], although they are expressed in ontology languages that are, in theory, not tractable in parallel. On the other hand, there are ontologies for which materialization performance does not improve by parallelism and our experiments in Section 6 confirm this. The current theoretical work of parallel ontology reasoning can hardly explain this. While one can try out different parallel implementations to see whether an ontology can be handled by (one of) them efficiently, we study the problem of parallel tractability in theory and identify properties that make an ontology tractable in parallel. These properties can further be used to optimize parallel algorithms and to guide ontology engineers in creating ontologies for which parallel tractability can be guaranteed theoretically.

According to Motik et al. [4], many real large-scale ontologies are essentially expressed in ontology languages that can be rewritten into datalog rules. We follow this argumentation and focus on such datalog rewritable ontology languages in this paper. The main target of this paper is to identify classes of datalog rewritable ontologies such that materialization over these ontologies is tractable in parallel, i.e., falls into the parallel complexity class NC [13]. This complexity class consists of problems that can be solved efficiently in parallel. To show that a problem is in NC, one can give an *NC algorithm* that handles this problem using parallel computation [13]. An NC algorithm is required to terminate in parallel poly-logarithmic time. However, current materialization algorithms of datalog rewritable ontology languages (e.g., the core algorithm used in RDFox [4]) are not NC algorithms since they are designed for general datalog programs and have PTime-complete complexity. Thus, we first give NC algorithms that perform materialization and then identify the corresponding classes of datalog rewritable ontologies (called *parallel tractability classes*) that can be handled by these NC algorithms. To make the proposed NC algorithms practical, we also discuss how to optimize and implement them.

In the practical part of this work, we study specific datalog rewritable ontology languages. We first focus on the ontology language DL-Lite [15] to clarify how to apply the above NC algorithms to study parallel tractability of ontology materialization. We show that DL-Lite$_{\text{core}}$ and DL-Lite$_{\mathcal{R}}$ are tractable in parallel. We then study the ontology language Description Horn Logic (DHL) [16], which is the intersection of datalog and OWL in terms of expressivity. We give a case of a DHL ontology where materialization can hardly be parallelized. Based on the analysis of this case, we propose to restrict the usage of DHL such that materialization over the restricted ontologies can be handled by the proposed NC algorithms. We further extend the results to an extension of DHL that also allows complex role inclusion axioms. Finally, we analyze the well-known benchmark, LUBM, and the real-world dataset, YAGO, and show that these ontologies satisfy the proposed restrictions and, hence, belong to the parallel tractability classes. We implement a system based on an optimized NC algorithm for DHL materialization,

which we compare to the state-of-the-art reasoner RDFox [4]. The experimental results show that the optimizations proposed in this paper result in a better performance on ontologies that are tractable in parallel compared to RDFox.

The remainder of the paper is organized as follows. In Section 2, we introduce some basic notions. We then give two NC algorithms for ontology materialization in Section 3. We study parallel tractability of materialization in DL-Lite, DHL and an extension of DHL in Section 4. In Section 5, we discuss how to optimize and implement the given NC algorithms. We analyze the LUBM and the YAGO ontologies and evaluate our implementation in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8. Detailed proofs of the theorems and lemmas in this paper are given in the appendix.

2. Background Knowledge

In this section, we introduce some basic notions that are needed to introduce our approach.

2.1. Datalog

We discuss the main issues in this paper using standard datalog notions. In datalog [17], a *term* is a variable or a constant. An *atom* A is defined by $A \equiv p(t_1, \ldots, t_n)$ where p is a *predicate* (or *relational*) name, $t_1, \ldots, t_n$ are terms, and n is the arity of p. If all the terms in an atom A are constants, then A is called a *ground atom*. A datalog *rule* is of the form: '$B_1, \ldots, B_n \rightarrow H$',[2] where H is referred to as the *head atom* and $B_1, \ldots, B_n$ the *body atoms*. Each variable in the head atom of a rule must occur in at least one body atom of the same rule. A *fact* is a rule of the form '$\rightarrow H$', i.e., a rule with an empty body and the head H being a ground atom. A datalog program $P = \langle R, \mathbf{I} \rangle$ consists of a finite set of rules R and a finite set of facts $\mathbf{I}$. A *substitution* θ is a partial mapping of variables to constants. For an atom A, $A\theta$ is the result of replacing each variable x in A with $\theta(x)$ if the latter is defined. We call θ a *ground substitution* if each defined $A\theta$ is a ground atom. A *ground instantiation* of a rule is obtained by applying a ground substitution on all the terms in this rule with respect to a finite set of constants occurring in P. Furthermore the ground instantiation of P, denoted by P^*, consists of all ground instantiations of rules in P. The predicates occurring only in the body of some rules are called *EDB predicates*, while the predicates that may occur as head atoms are called *IDB predicates*.

2.2. DHL and DL-Lite

In what follows, we use **CN**, **RN** and **IN** to denote three disjoint countably infinite sets of *concept names, role names*, and *individual names* respectively. The set of roles is defined as $\mathbf{R} := \mathbf{RN} \cup \{R^- | R \in \mathbf{RN}\}$ where R^- is the *inverse role* of R. For ease of discussion, we focus on the *simple forms* of axioms shown in the left column of Table 1. These simple forms can be obtained by using well-known *structural transformation* techniques [18, 19].

DHL (short for *description horn logic*) [16] is introduced as an intersection of description logic (DL) and datalog in terms of expressivity. We define a DHL ontology $\mathcal{O}$ as a triple: $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, where $\mathcal{T}$ denotes the TBox containing axioms of the forms (T1-T3); $\mathcal{R}$ is the RBox that is a set of axioms of the forms (R1-R3); $\mathcal{A}$ is the ABox containing *assertions* of the forms (A1) and (A2). In an axiom of either of the forms (T1-T3 and R1-R3), concepts $A_{(i)}$ and B are

[2] In datalog rules, a comma represents a Boolean conjunction '$\wedge$'.

Table 1: Axioms and corresponding datalog rules

	Axioms	Datalog Rules
(T1)	$A \sqsubseteq B$	$A(x) \rightarrow B(x)$
(T2)	$A_1 \sqcap A_2 \sqsubseteq B$	$A_1(x), A_2(x) \rightarrow B(x)$
(T3)	$\exists R.A \sqsubseteq B$	$R(x, y), A(y) \rightarrow B(x)$
(T4)	$A \sqsubseteq \exists R$	$A(x) \rightarrow R(x, o_R^A)$
(R1)	$S \sqsubseteq R$	$S(x, y) \rightarrow R(x, y)$
(R2)	$S \sqsubseteq R^-$	$S(x, y) \rightarrow R(y, x)$
(R3)	$R \circ R \sqsubseteq R$	$R(x, y), R(y, z) \rightarrow R(x, z)$
(R4)	$R_1 \circ R_2 \sqsubseteq R$	$R_1(x, y), R_2(y, z) \rightarrow R(x, z)$
(A1)	$A(a)$	$A(a)$
(A2)	$R(a, b)$	$R(a, b)$

either concept names, the *top concept* ($\top$) or the *bottom concept* ($\bot$); R and $S_{(i)}$ are roles in $\mathbf{R}$. For an axiom of the form $A \sqsubseteq \forall R.B$ that is also allowed in DHL, we only consider its equivalent form $\exists R^-.A \sqsubseteq B$.

DHL is related with other ontology languages. First, DHL is essentially a fragment of the description logic Horn-$\mathcal{SHOIQ}$ with disallowing *nominals*, *number restrictions* and right-hand side *existential restrictions* ($A \sqsubseteq \exists R.B$). Second, the expressivity of DHL covers that of RDFS to some extent [16]. Reasoning with RDFS ontologies is `NP`-complete [20] and, thus, is not tractable in parallel. However, by applying some simplifications and restrictions, RDFS ontologies can be expressed in DHL [16], which has `PTime`-complete complexity for materialization.

In the initial work of DHL [16], *complex role inclusion axioms* (complex RIAs) of the form $R_1 \circ \ldots \circ R_n \sqsubseteq R$ are not considered, although they can be naturally transformed into datalog rules. In this paper, we also consider an extension of DHL (denoted by DHL($\circ$)) that allows complex RIAs. Since a complex RIA can be transformed to several axioms of form (R4), we then require that an RBox $\mathcal{R}$ of a DHL($\circ$) ontology can contain axioms of the forms (R1-R4). Note that (R3) is actually a special case of (R4).

DL-Lite is a group of ontology languages designed for highly-efficient query answering over knowledge bases and underpins the OWL profile OWL QL [15]. DL-Lite$_{\text{core}}$ and DL-Lite$_{\mathcal{R}}$ are the two basic fragments of DL-Lite. DL-Lite$_{\text{core}}$ requires that a TBox only contains axioms of the forms (T1) $A \sqsubseteq B$, (T2) $A_1 \sqcap A_2 \sqsubseteq B$ where $B \equiv \bot$, (T3) $\exists R.A \sqsubseteq B$ where $A \equiv \top$ or $B \equiv \bot$, and (T4) $A \sqsubseteq \exists R$, an ABox contains *assertions* of the forms (A1) and (A2), and RBoxes are not allowed. DL-Lite$_{\mathcal{R}}$ is an extension of DL-Lite$_{\text{core}}$, which allows involving RBoxes that contain axioms of the forms (R1-R2). If not specially specified, *a DL-Lite ontology denotes a DL-Lite$_{\mathcal{R}}$ ontology in the following paragraphs*.

2.3. Ontology Materialization via Datalog Programs

An ontology expressed in DHL, DHL($\circ$), DL-Lite$_{\text{core}}$ or DL-Lite$_{\mathcal{R}}$ can be transformed into a datalog program (see the corresponding rules in the right column of Table 1). Note that an axiom of form (T4) $A \sqsubseteq \exists R$ should be transformed into a first-order logic rule of the form $A(x) \rightarrow \exists y(R(x, y))$, where y is called a *free variable* (it is not occurring in the body atom $A(x)$). In this work, such a free variable y is eliminated via Skolemization into a fresh individual o_R^A that corresponds to the concept A and the role R. The elimination of free variables does not influence the result of materialization [15].

In what follows, for an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, we use $P = \langle R, \mathbf{I} \rangle$ to represent the corresponding datalog program where R is the set of rules obtained by transforming the axioms in $\mathcal{T}$ and $\mathcal{R}$ and $\mathbf{I}$ is the set of facts that are directly copied from the assertions in $\mathcal{A}$. Further, we use $R_1 \sqsubseteq_* R_2$ to denote the smallest transitive reflexive relation between roles such that $R_1 \sqsubseteq R_2 \in \mathcal{R}$ implies $R_1 \sqsubseteq_* R_2$ and $R_1^- \sqsubseteq_* R_2^-$. We omit the reference to $\mathcal{R}$ if it is clear from the context. In this paper, we also use the notion of *simple role*, which was initially proposed to restrict the usage of highly expressive ontology languages [21]. Specifically, a role $S \in \mathbf{R}$ is *simple* if, (1) it has no subrole (including S) occurring on the right-hand side of axioms of the forms (R3) and (R4); (2) S^- is simple.

Based on the above representations, ontology materialization corresponds to the evaluation of datalog programs. Specifically, given a datalog program $P = \langle R, \mathbf{I} \rangle$, we set

$$T_R(\mathbf{I}) = \{H\theta \mid \forall B_1, \ldots, B_n \rightarrow H \in R, B_i\theta \in \mathbf{I}, 1 \leq i \leq n\},$$

where θ is some substitution; further let $T_R^0(\mathbf{I}) = \mathbf{I}$ and $T_R^i(\mathbf{I}) = T_R^{i-1}(\mathbf{I}) \cup T_R(T_R^{i-1}(\mathbf{I}))$ for each $i > 0$. The smallest integer n such that $T_R^n(\mathbf{I}) = T_R^{n+1}(\mathbf{I})$ is called *stage*, and *materialization* refers to the computation of $T_R^n(\mathbf{I})$ with respect to R and $\mathbf{I}$. $T_R^n(\mathbf{I})$ is also called the *fixpoint* and denoted by $T_R^\omega(\mathbf{I})$. We say that an atom A is *derivable* or can be *derived* with respect to the datalog program $P = \langle R, \mathbf{I} \rangle$ if $A \in T_R^\omega(\mathbf{I})$. In this paper, we consider the data complexity of materialization, i.e., we *assume that the rule set R is fixed* for any class of datalog programs.

2.4. The Complexity Class NC

The parallel complexity class NC, known as Nick's Class [13], is studied by theorists as a parallel complexity class where each decision problem can be efficiently solved in parallel. Specifically, a decision problem in the NC class can be solved in poly-logarithmic time on a PRAM (parallel, random-access machine) with a polynomial number of processors. We also say that an NC problem can be solved in *parallel poly-logarithmic time*. Although the NC complexity class is a theoretical analysis tool, it has been shown that many NC problems can be solved efficiently in practice [13].

From the perspective of implementations, NC problems are also highly feasible in parallel for other parallel models like BSP [22] and MapReduce [23]. The NC complexity class was originally defined as a class of decision problems. Since we study the problem of materialization, we do not require in this work that a problem is a decision problem in NC. In addition, since many parallel reasoning systems (see related work in Section 7) are implemented on shared-memory platforms, we study all the issues in this work by assuming that the running machines are in shared-memory configurations.

3. Parallel Tractability of Datalog Programs

Our target is to find for which kinds of ontologies (not ontology languages) materialization is tractable in parallel. We consider data complexity, i.e., we consider classes of datalog programs which share a fixed rule set. The data complexity of the materialization problem in this case is PTime-complete, which is considered to be inherently sequential in the worst case [13]. In other words, the materialization problem of datalog programs cannot be solved in parallel poly-logarithmic time unless P=NC. Thus, we say that *materialization for a class of datalog programs is tractable in parallel if there exists an algorithm that handles this class of datalog programs and runs in parallel poly-logarithmic time. Such an algorithm is also called an NC algorithm.* In this section, we identify such classes of datalog programs by studying different NC algorithms.

3.1. Parallel Tractability Classes

We first give the following definition for a class of datalog programs that is tractable in parallel, i.e., an NC algorithm exists for handling each datalog program in this class.

Definition 1 (Parallel Tractability Class)**.** *Given a class $\mathcal{D}$ of datalog programs sharing the same rule set, we say that $\mathcal{D}$ is a class of datalog programs with* parallel tractability*, short a* PTD class*, if there exists an NC algorithm that performs sound and complete materialization for each datalog program in $\mathcal{D}$. The corresponding class of ontologies of $\mathcal{D}$ is called a class of ontologies with parallel tractability, short a PTO class.*

Suggested revision: (Parallel Tractability Class). Let R be a set of Datalog rules. Materialization w.r.t. to R is tractable in parallel, if, for every set of facts $\boldsymbol{I}$, there exists an NC algorithm that performs sound and complete materialization for $P = \langle R, \boldsymbol{I}\rangle$; the set $\mathcal{D}$ of all such Datalog programs P is called a PTD class*. The corresponding class of ontologies of $\mathcal{D}$ is called a class of ontologies with parallel tractability, short a PTO class.*

According to the above definition, if we find an NC algorithm A for datalog materialization, then we can identify a PTD class $\mathcal{D}_{\mathsf{A}}$, which is the class of all datalog programs that can be handled by A. However, current materialization algorithms of datalog rewritable ontology languages (e.g., the core algorithm used in RDFox [4]) are not NC algorithms due to their PTime-complete complexity, since they are designed for handling general datalog programs. Thus, we proceed to devise specific NC algorithms. In the following, we first give a parallel materialization algorithm that works for general datalog programs. We then restrict this algorithm to an NC version and identify the target PTD class.

3.2. Materialization Graph

In order to give a parallel materialization algorithm, we introduce the notion of *materialization graphs*, which facilitates the analysis of the given algorithm.

Definition 2 (Materialization Graph)**.** *A* materialization graph*, with respect to a datalog program $P = \langle R, \boldsymbol{I}\rangle$, is a directed acyclic graph denoted by $\mathcal{G} = \langle V, E\rangle$ where,*

- *V is the node set and $V \subseteq T_R^\omega(\boldsymbol{I})$;*
- *E is the edge set and $E \subseteq T_R^\omega(\boldsymbol{I}) \times T_R^\omega(\boldsymbol{I})$.*

For each edge of the form $e(v_1, v_2)$ where v_1 and v_2 are two nodes, we say that the node v_1 is the parent (node) of v_2 and the node v_2 is the child (node) of v_1. Further, $\forall H, B_1, \ldots, B_n \in V$, $\mathcal{G}$ satisfies the following conditions:

- *H has a parent node or a child node;*
- *if H has an in-degree of 0, then H is an original fact of P;*
- *$B_1, \ldots, B_n \rightarrow H \in P^*$ is satisfied if $e(B_1, H), \ldots, e(B_n, H) \in E$ and $B_1, \ldots, B_n$ are all the parents of H.*

A materialization graph $\mathcal{G}$ is a complete materialization graph*, when $\mathcal{G}$ contains all ground atoms in $T_R^\omega(\boldsymbol{I})$.*

Suggested revision: (Materialization Graph). A materialization graph*, with respect to a datalog program $P = \langle R, \boldsymbol{I}\rangle$, is a directed acyclic graph denoted by $\mathcal{G} = \langle V, E\rangle$ where,*

- *V is the node set and $V \subseteq T_R^\omega(\boldsymbol{I})$;*
- *E is the edge set and $E \subseteq T_R^\omega(\boldsymbol{I}) \times T_R^\omega(\boldsymbol{I})$.*

For each edge of the form $e(v_1, v_2)$ where v_1 and v_2 are two nodes, we say that the node v_1 is the parent (node) of v_2 and the node v_2 is the child (node) of v_1.

Further, for each $H, B_1, \ldots, B_n \in V$, $\mathcal{G}$ satisfies the following conditions:

- *if $e(B_1, H), \ldots, e(B_n, H) \in E$ and $B_1, \ldots, B_n$ are all the parents of H, then $B_1, \ldots, B_n \rightarrow H \in P^*$;*
- *if H has an in-degree of 0, then H is an original fact of P.*

A materialization graph $\mathcal{G}$ is a complete materialization graph*, when $\mathcal{G}$ contains all ground atoms in $T_R^\omega(\boldsymbol{I})$.*

For some derived atom H, there may exist several rule instantiations where H occurs as a head atom. This also means that H can be derived in different ways. The condition in the definition above results in only one way of deriving H being described by a materialization graph. Suppose $\mathcal{G}$ is a materialization graph such that the nodes with in-degree 0 are the original facts in $\mathbf{I}$. The size of $\mathcal{G}$, denoted by $|\mathcal{G}|$, is the number of nodes in $\mathcal{G}$. The depth of $\mathcal{G}$, denoted by $\mathtt{depth}(\mathcal{G})$, is the maximal length of a path in $\mathcal{G}$. We next give an example of a materialization graph.

Example 1. *Consider a DHL($\circ$) ontology $\mathcal{O}_{ex_1}$ where the TBox is $\{\exists R.A \sqsubseteq A\}$, the RBox is $\{S \circ R \sqsubseteq R\}$ and the ABox is $\{A(b), R(a_1, b), S(a_i, a_{i-1})\}$ for $2 \leq i \leq k$ and k an integer greater than 2. The corresponding datalog program of this ontology is $P_{ex_1} = \langle R, \boldsymbol{I} \rangle$ where $\boldsymbol{I}$ contains all the assertions in the ABox and R contains the two rules '$R(x, y), A(y) \rightarrow A(x)$' and '$S(x, y), R(y, z) \rightarrow R(x, z)$'. The graph in Figure 1 is a materialization graph with respect to P_{ex_1}, denoted by $\mathcal{G}_{ex_1}$. The nodes with in-degree 0 are the original facts in $\boldsymbol{I}$; each of the other nodes corresponds to a ground instantiation of some rule. For example, the node $A(a_k)$ corresponds to the ground rule instantiation '$R(a_k, b), A(b) \rightarrow A(a_k)$'. The size of this materialization graph is the number of nodes, that is $3k$. The depth of $\mathcal{G}_{ex_1}$ is k.*

The set of nodes in a complete materialization graph is actually the result of materialization. Thus, the procedure of materialization can be transformed into the construction of a complete materialization graph. In the remainder, we only consider complete materialization graphs and do not distinguish the term from the notion of 'materialization graphs'. It should also be noted that there may exist several materialization graphs for a datalog program.

3.3. A Basic Parallel Algorithm

In this part, we propose a parallel algorithm (denoted by Algorithm A_{bsc}) that constructs a materialization graph for a given datalog program. Our computational model is a PRAM (parallel, random-access machine) that is mostly used to analyze parallel complexity. This model allows us to introduce *the parallel assumption*: for any datalog program $P = \langle R, \mathbf{I} \rangle$, any substitution of some atom and any rule instance in P^* can be mapped to a unique memory location; further, a one-to-one relation can be established between processors and rule instances. Under this assumption, a processor can check the applicability of its corresponding rule instance and

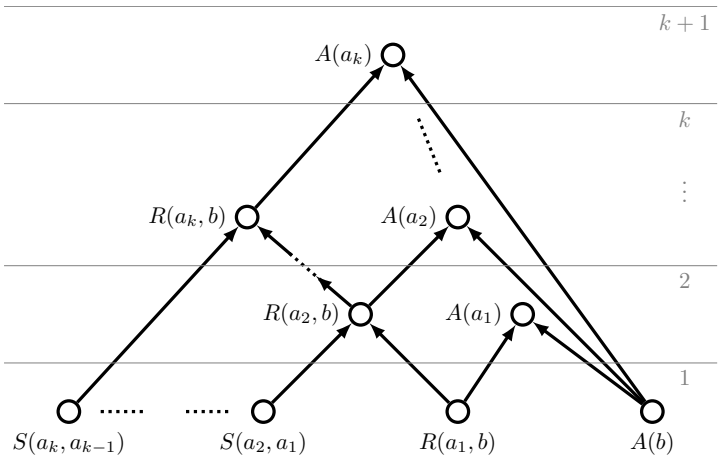

Figure 1: An example materialization graph

access the state of an atom occurring in this rule instance in constant time. Algorithm $\mathsf{A}_{\mathrm{bsc}}$ is then given as follows.

`Algorithm` $\mathsf{A}_{\mathtt{bsc}}$. Given a datalog program $P = \langle R, \mathbf{I}\rangle$, the algorithm returns a materialization graph $\mathcal{G}$ of P. Suppose we have $|P^*|$ processors, and each rule instantiation in P^* is assigned to one processor. Initially $\mathcal{G}$ is empty. The following three steps are then performed:

(*Step 1*) Add all facts in $\mathbf{I}$ to $\mathcal{G}$.

(*Step 2*) For each rule instantiation $B_1, \ldots, B_n \rightarrow H$, if the body atoms are all in $\mathcal{G}$ while H is not in $\mathcal{G}$, the corresponding processor adds H to $\mathcal{G}$ and creates edges pointing from $B_1, \ldots, B_n$ to H.

(*Step 3*) If no processor can add more nodes and edges to $\mathcal{G}$, terminate, otherwise continue with Step 2. □

Example 2. *We consider the datalog program P_{ex_1} in Example 1 again, and perform Algorithm A_{bsc} on it. Initially, all the facts $(A(b), R(a_1, b), S(a_2, a_1), \ldots, S(a_k, a_{k-1}))$ are added to the result $\mathcal{G}_{ex_1}$ (Step 1). Then in different iterations of Step 2, the remaining nodes are added to $\mathcal{G}_{ex_1}$ by different processors. For example a processor p is allocated a rule instantiation '$R(a_2, b), A(b) \rightarrow A(a_2)$'. Then, processor p adds $A(a_2)$ to $\mathcal{G}_{ex_1}$ after it checks that $A(b)$ and $R(a_2, b)$ are in $\mathcal{G}_{ex_1}$. Algorithm A_{bsc} halts when $A(a_k)$ has been added to $\mathcal{G}_{ex_1}$ (Step 3).*

Lemma 1 shows the correctness of Algorithm $\mathsf{A}_{\mathrm{bsc}}$ and that, for any datalog program P, Algorithm $\mathsf{A}_{\mathrm{bsc}}$ always constructs a materialization graph with the minimum depth among all the materialization graphs of P. The detailed proofs of Lemma 1 and other lemmas and theorems can be found in the appendix.

Lemma 1. *Given a datalog program $P = \langle R, \boldsymbol{I}\rangle$, we have*

1. *Algorithm A_{bsc} halts and returns a materialization graph $\mathcal{G}$ of P;*

2. *$\mathcal{G}$ has the the minimum depth among all the materialization graphs of P.*

Proof sketch. This lemma can be proved by performing an induction on $T_R^\omega(\mathbf{I})$. The stage (see the related contents in Section 2) of P is the lower-bound of the depth of the materialization graphs. Based on the previous induction, one can further check that, for the materialization graph $\mathcal{G}$ constructed by Algorithm A_{bsc}, its depth is equal to the depth of the stage. □

We now discuss the parallel complexity of Algorithm A_{bsc}. Given a class of datalog programs $\mathbb{P}$ where a rule set is shared for each datalog program $P = \langle R, \mathbf{I}\rangle$ in $\mathbb{P}$. Let e, v and r represent the maximum arity of any predicate in $\mathbf{I}$, the maximum number of variables in any datalog rule, and the number of datalog rules respectively. We then have that the number of constants is at most $|\mathbf{I}|e$, and the number of all possible rule instances in P^* is at most $r(|\mathbf{I}|e)^v$. Note that e, v and r depend only on the rule set R and not on the fact set $\mathbf{I}$. Thus, the memory space for storing the atoms and the rule instances is polynomial in the size of $\mathbf{I}$. This also means that the number of processors is polynomially bounded. The computing time of Step 1 and Step 3 occupy constant time (denoted by c_1) because of parallelism. Since Algorithm A_{bsc} works under the parallel assumption, one iteration of Step 2 also costs constant time (denoted by c_2). Thus, the whole computing time of Algorithm A_{bsc} turns out to be $c_1 + \ell \cdot c_2$ where ℓ denotes the number of iterations of Step 2.

An NC algorithm should meet two requirements: first, it works on a polynomial number of processors; second, it halts in poly-logarithmic time. As discussed above, Algorithm A_{bsc} meets the first requirement. If we want to restrict Algorithm A_{bsc} to be an NC algorithm, we can make the number of iterations of Step 2 to be poly-logarithmically bounded. We use the symbol ψ to denote a poly-logarithmically bounded function. For any datalog program, if we restrict the number of iterations of Step 2 to be bounded by $\psi(|\mathbf{I}|)$, the computing time of Algorithm A_{bsc} is $c_1 + \psi(|\mathbf{I}|) \cdot c_2$. With this restriction, Algorithm A_{bsc} is an NC algorithm denoted by $\mathsf{A}^{\psi}_{\text{bsc}}$.

Based on $\mathsf{A}^{\psi}_{\text{bsc}}$, we can identify a class of datalog programs $\mathcal{D}_{\mathsf{A}^{\psi}_{\text{bsc}}}$ such that all the datalog programs in it can be handled by $\mathsf{A}^{\psi}_{\text{bsc}}$. It is obvious that $\mathcal{D}_{\mathsf{A}^{\psi}_{\text{bsc}}}$ is a PTD class. We use the following theorem to further show that this class can be captured based on the materialization graphs of the datalog programs in $\mathcal{D}_{\mathsf{A}^{\psi}_{\text{bsc}}}$.

Theorem 1. *For any datalog program $P = \langle R, \boldsymbol{I}\rangle$, $P \in \mathcal{D}_{\mathsf{A}^{\psi}_{bsc}}$ iff P has a materialization graph whose depth is upper-bounded by $\psi(|\boldsymbol{I}|)$.*

Proof sketch. We can first prove that the number of iterations of Step 2 is actually the depth of the constructed materialization graph. This theorem then follows by considering Lemma 1. □

Consider Example 1 again. Let the integer k be a variable. We can get a class of datalog programs, denoted by $\mathbb{P}_{\text{ex}_1}$, where the rule set is $\{R(x,y), A(y) \rightarrow A(x), S(x,y), R(y,z) \rightarrow R(x,z)\}$ and the fact set varies according to k. The algorithm $\mathsf{A}^{\psi}_{\text{bsc}}$ is restricted in the sense that it cannot even work on the rather simple datalog program class $\mathbb{P}_{\text{ex}_1}$. Let $\mathcal{G}_{\text{ex}_1}$ be some materialization graph corresponding to some datalog program in $\mathbb{P}_{\text{ex}_1}$. It can be checked that $\texttt{depth}(\mathcal{G}_{\text{ex}_1}) = k$ for some k. This means that the depths of the materialization graphs are linearly bounded by k. On the other hand, the sizes of the datalog programs in $\mathbb{P}_{\text{ex}_1}$ are polynomial in k. Thus, for any ψ that is poly-logarithmically bounded, we can always find a k large enough such that $\mathsf{A}^{\psi}_{\text{bsc}}$ terminates without constructing a materialization graph for each datalog program in $\mathbb{P}_{\text{ex}_1}$. However, there indeed exists an NC algorithm that can handle $\mathbb{P}_{\text{ex}_1}$. We discuss this in the next part.

3.4. Optimizing Algorithm A_{bsc} via Single-Way Derivability

In this part, we optimize Algorithm A_{bsc} such that P_{ex_1} can be handled. Based on the optimized variant of Algorithm A_{bsc}, we can identify another PTD class.

We discuss our optimization based on a specific case in Example 3. We find that, in this kind of case, the construction of a materialization graph can be accelerated.

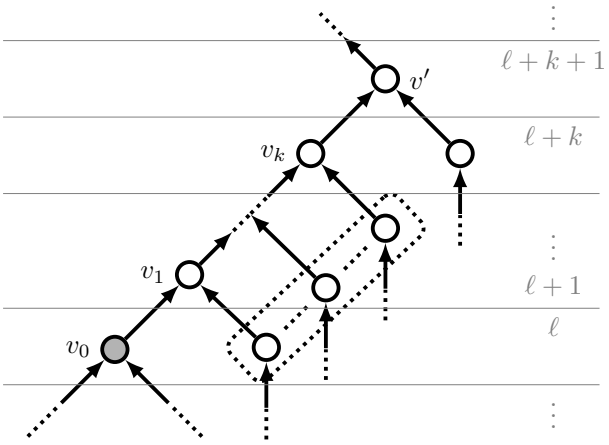

Figure 2: A partial materialization graph

Example 3. *Consider the snapshot of Algorithm A_{bsc} in Figure 2. A materialization graph $\mathcal{G}$ is being constructed for some datalog program $\langle R, \boldsymbol{I}\rangle$. The nodes in the dashed box denote the ones that have been added to $\mathcal{G}$. In this snapshot, v_0 has been newly added to $\mathcal{G}$ in the ℓ^{th} ($\ell \geq 1$) iteration. Each of the nodes v_i ($1 \leq i \leq k$) has at most one parent node not in $\mathcal{G}$, while v' has two parent nodes not in $\mathcal{G}$. All of the nodes v_i ($1 \leq i \leq k$) and v' would be added to $\mathcal{G}$ afterwards.*

In Example 3, v_k would be added to $\mathcal{G}$ after *at least k* iterations by performing Algorithm A_{bsc}. We can check that each edge (v_{i-1}, v_i) $(1 \leq i \leq k)$ in this stage adheres to the condition: *if the parent node v_{i-1} is derivable, the child node v_i is derivable* (this is because for a node v_i, the node v_{i-1} is the only parent node that has not been added to $\mathcal{G}$). We call this condition the *single-way derivability condition*, which intuitively says that the derivability of a node only depends on one of its parent nodes. Observe that node v_k is reachable from node v_0 through the path $\tau = (v_0, v_1, \ldots, v_k)$ where each edge (v_{i-1}, v_i) $(1 \leq i \leq k)$ satisfies the single-way derivability condition. We call such a path a *single-way derivable* (SWD) path. Further, the starting node v_0 in τ has been added to $\mathcal{G}$; in other words, v_0 is derivable. Thus, all of the nodes v_i $(1 \leq i \leq k)$ can be added to $\mathcal{G}$ immediately. Based on the above idea, we optimize Algorithm A_{bsc} by checking whether such an SWD path exits for some node v. If there is an SWD path for node v, this node can be added to $\mathcal{G}$ due to the single-way derivability condition.

For Example 3, there exists an SWD path $(v_0, \ldots, v_i)$ for each node v_i $(1 \leq i \leq k)$, hence, all of these nodes can be added to $\mathcal{G}$ right after v_0. On the other hand, node v' has no SWD path since it has two parent nodes not in $\mathcal{G}$.

We next discuss how to determine the existence of SWD paths. Note that, SWD paths require us to describe the reachability between two nodes. To this extent, we use a binary transitive

relation $\texttt{rch} \subseteq T_R^\omega(\mathbf{I}) \times T_R^\omega(\mathbf{I})$, e.g., $\texttt{rch}(v_1, v_2)$ means that v_2 is reachable from v_1. In each iteration of Step 2 in Algorithm A_{bsc}, we further compute as rch relation (denoted by S_{rch}) by performing the following process:

(†) *For each rule instantiation of the form $B_1, .., B_i, .., B_n \rightarrow H$ such that H has not been added to $\mathcal{G}$:*

1. *if the all body atoms $B_1, \ldots, B_n$ have been added to $\mathcal{G}$, put $\texttt{rch}(B_1, H), \ldots, \texttt{rch}(B_n, H)$ in S_{rch};*
2. *if B_i is the only node in the body that has not been added to $\mathcal{G}$, put $\texttt{rch}(B_i, H)$ in S_{rch}.* □

We then compute the transitive closure (denoted by S_{rch}^*) with respect to S_{rch}. Based on the transitive closure, we can perform the following optimization: for a node v, if there is a relation $\texttt{rch}(v', v) \in S_{rch}^*$ such that v' has been added to $\mathcal{G}$ and v has an SWD path, then v can be added to $\mathcal{G}$. The following algorithm applies this optimization strategy.

`Algorithm` OPT. The algorithm requires two inputs: a datalog program $P = \langle R, \mathbf{I} \rangle$ and a (partial) materialization graph $\mathcal{G}$ that is being constructed from P. The following steps are performed:

(**i**) Compute the rch relation S_{rch} by following the above process (see (†)).

(**ii**) Compute the transitive closure S_{rch}^* of S_{rch}.

(**iii**) Update $\mathcal{G}$ as follows: for any $\texttt{rch}(B_i, H) \in S_{rch}$ that corresponds to '$B_1, .., B_i, .. , B_n \rightarrow H$' and there exists a node B' such that $\texttt{rch}(B', B_i) \in S_{rch}^*$ and B' is in $\mathcal{G}$; if H is not in $\mathcal{G}$ or H is in $\mathcal{G}$ but has no parent pointing to it, add H and B_i (if B_i is not in $\mathcal{G}$) to $\mathcal{G}$, and create the edges $e(B_1, H), \ldots, e(B_n, H)$ in $\mathcal{G}$. Do nothing for other statements $\texttt{rch}(B_j, H) \in S_{rch}$. □

It is well known that there is an NC algorithm for computing the transitive closure [24]. Based on this result and Algorithm OPT, we propose an optimized variant of Algorithm A_{bsc}:

`Algorithm` A_{opt}. Given a datalog program $P = \langle R, \mathbf{I} \rangle$, the algorithm returns a materialization graph $\mathcal{G}$ of P. Suppose we have $|P^*|$ processors, and each rule instantiation in P^* is assigned to one processor. Initially $\mathcal{G}$ is empty. The following steps are then performed:

(*Step 1*) Add all facts in $\mathbf{I}$ to $\mathcal{G}$.

(*Step 2*) Compute S_{rch} by performing (**i**) in Algorithm OPT; use an NC algorithm to compute the transitive closure S_{rch}^* (see (**ii**) in Algorithm OPT); update $\mathcal{G}$ by performing (**iii**) in Algorithm OPT.

(*Step 3*) If no node has been added to $\mathcal{G}$ (in Step 2), terminate, otherwise iterate Step 2. □

It should be noted that there has to be an SWD path for any derivable node in some iteration when performing Algorithm A_{opt}. The following lemma shows the correctness of Algorithm A_{opt}.

Lemma 2. *Given a datalog program $P = \langle R, \boldsymbol{I} \rangle$, A_{opt} halts and outputs a materialization graph $\mathcal{G}$ of P.*

Example 4. *We perform Algorithm A_{opt} on the datalog program P_{ex_1} in Example 1. Initially, $R(a_1, b)$ is in the materialization graph $\mathcal{G}_{ex_1}$. In the first iteration of Step 2, all the rule instantiations are in two kinds of forms: '$R(a_i, b), A(b) \rightarrow A(a_i)$' and '$S(a_i, a_{i-1}), R(a_{i-1}, b) \rightarrow R(a_i, b)$', $2 \leq i \leq k$, S_{rch} is the set $\{\mathtt{rch}(R(a_{i-1}, b), R(a_i, b)) \mid 2 \leq i \leq k\} \cup \{\mathtt{rch}(R(a_i, b), A(a_i)) \mid 1 \leq i \leq k\}$. In the transitive closure of S_{rch}, one can check that $\mathtt{rch}(R(a_1, b), R(a_i, b))$, $\mathtt{rch}(R(a_1, b), A(a_i)) \in S^*_{rch}, 2 \leq i \leq k$. Thus, $R(a_i, b)$ and $A(a_i)$, $2 \leq i \leq k$, can all be added to $\mathcal{G}_{ex_1}$ in the first iteration of Step 2.*

We now analyse the parallel complexity of Algorithm A_{opt}. We use the same symbols e, v and r that are used for analyzing the complexity of Algorithm A_{bsc} (see Section 3.3). Similarly to the analysis for Algorithm A_{bsc}, it can be checked that the number of processors is polynomial in $|\mathbf{I}|$, i.e., $r(|\mathbf{I}|e)^v$. We now consider the computing time of Algorithm A_{opt}. It is obvious that Step 1 and Step 3 occupy constant time (denoted by c_1). In Step 2, the phases of computing S_{rch} and updating $\mathcal{G}$ also take constant time (denoted by c_2) under the parallel assumption. Recall that the number of constants is at most $|\mathbf{I}|e$ (see Section 3.3). Let w and p represent the maximum arity of any predicate and the number of predicates, respectively. We have that the number of all possible substitutions of atoms is at most $p(|\mathbf{I}|e)^w$. The relation S_{rch} consists of pairs of the atoms. Thus, the size of S_{rch} is bounded by $p^2(|\mathbf{I}|e)^{2w}$. It can be checked that the computing time of the NC algorithm for computing the transitive closure of S_{rch} is poly-logarithmic in the size of $\mathbf{I}$, more precisely, it is upper bounded by $\log^2(p^2(|\mathbf{I}|e)^{2w})$ $(= 4\log^2(p) + 4w^2\log^2(|\mathbf{I}|e) + 8w\log(p)\log(|\mathbf{I}|e))$. We now have that the total computing time of Algorithm A_{opt} is $c_1 + \ell c_3 + \ell c_4 \log(|\mathbf{I}|e) + \ell c_5 \log^2(|\mathbf{I}|e)$ where $c_3 = c_2 + 4\log^2(p)$, $c_4 = 8w\log(p)$, $c_5 = 4w^2$ and ℓ denotes the number of iterations of Step 2.

Algorithm A_{opt} can be restricted to an NC algorithm analogously to the process for Algorithm A_{bsc}. That is the number of iterations of Step 2 is poly-logarithmically bounded, i.e., it is bounded by $\psi(|\mathbf{I}|)$ where ψ is a poly-logarithmic function. In this way, the computing time of Algorithm A_{bsc} turns out to be $c_1 + c_3\psi(|\mathbf{I}|) + c_4\psi(|\mathbf{I}|)\log(|\mathbf{I}|e) + c_5\psi(|\mathbf{I}|)\log^2(|\mathbf{I}|e)$, which is still poly-logarithmical in the size of $\mathbf{I}$. We denote the NC variant of Algorithm A_{opt} by $\mathsf{A}^{\psi}_{\text{opt}}$.

Based on $\mathsf{A}^{\psi}_{\text{opt}}$, we can identify a PTD class $\mathcal{D}_{\mathsf{A}^{\psi}_{\text{opt}}}$. Further, we have the following corollary, which implies that $\mathsf{A}^{\psi}_{\text{opt}}$ performs better than $\mathsf{A}^{\psi}_{\text{bsc}}$ in terms of computing time.

Corollary 1. *For any poly-logarithmically bounded function ψ, we have that $\mathcal{D}_{\mathsf{A}^{\psi}_{bsc}} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

Proof sketch. Suppose $P = \langle R, \mathbf{I}\rangle \in \mathcal{D}_{\mathsf{A}^{\psi}_{\text{bsc}}}$. According to Theorem 1, the depth of the materialization graph $\mathcal{G}$ constructed by $\mathsf{A}^{\psi}_{\text{bsc}}$ is upper-bounded by $\psi(|\mathbf{I}|)$. It is obvious that the number of nodes in each path of $\mathcal{G}$ is also upper-bounded by $\psi(|\mathbf{I}|)$. According to the optimization strategy applied in A_{opt}, if $\mathcal{G}$ can be constructed by A_{opt}, the number of iterations of A_{opt} has to be upper-bounded by $\psi(|\mathbf{I}|)$; if $\mathcal{G}$ is not the materialization graph constructed by A_{opt}, then there has to exist another materialization graph $\mathcal{G}'$ constructed by A_{opt} and $\mathcal{G}'$ has a smaller depth compared to $\mathcal{G}$. □

4. Parallel Tractability of Ontology Materialization in OWL

In this section, we study the issue of parallel tractability for materialization of DL-Lite and DHL (DHL(∘)) ontologies based on Algorithm A_{opt}. We show that, for any class $\mathbb{O}$ of DL-Lite$_{\text{core}}$ or DL-Lite$_{\mathcal{R}}$ ontologies, there exists a poly-logarithmically bounded function ψ such that Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ can handle materialization of the ontologies in $\mathbb{O}$. However, for DHL and

DHL(∘), there exist ontology classes such that Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ does not work. We illustrate the reason why Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ cannot always work by studying specific cases. Further, we propose to restrict the usage of DHL and DHL(∘) in order to achieve parallel tractability of materialization.

4.1. Materialization of DL-Lite Ontologies via Algorithm A_{opt}

In this part, we show how to use Algorithm A_{opt} to handle DL-Lite materialization and analyze its parallel tractability. Based on the analysis, we have that, for any DL-Lite$_{\text{core}}$ or DL-Lite$_{\mathcal{R}}$ ontology there always exists an SWD path for each atom of the form $A(a)$ or $R(a, b)$. In other words, all atoms of the forms $A(a)$ and $R(a, b)$ can be added to the constructed materialization graph in the first iteration of Step 2 by performing Algorithm A_{opt}.

In order to show how Algorithm A_{opt} handles DL-Lite materialization, we use the following example.

Example 5. *Given a DL-Lite ontology O_{ex_5} where the TBox and RBox contain the following axioms: $A \sqsubseteq B_1$, $A \sqsubseteq B_2$, $B_1 \sqcap B_2 \sqsubseteq \bot$, $\exists R.B_2 \sqsubseteq \bot$, $Q \sqsubseteq S^-$, $S \sqsubseteq R$; its ABox contains an assertion $Q(a, b)$. We denote the corresponding datalog program of O_{ex_5} by $P_{ex_5} = \langle \boldsymbol{R}, \boldsymbol{I} \rangle$, where $\boldsymbol{R}$ contains the rules that are transformed from the above axioms; $\boldsymbol{I}$ contains $Q(a, b)$ as the only fact. The unique materialization graph of P_{ex_5} is denoted by $\mathcal{G}_{ex_5}$ (see Figure 3).*

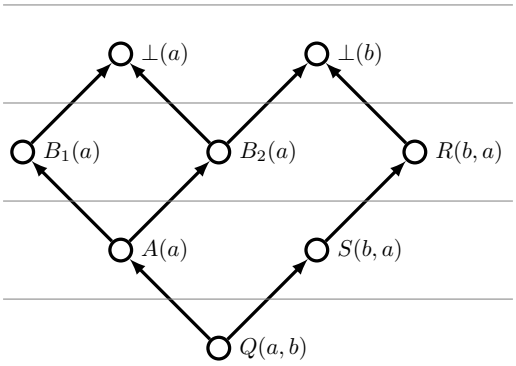

Figure 3: The materialization graph of O_{ex_5}

Consider performing Algorithm A_{opt} on the ontology O_{ex_5} in Example 5. (I) First, Algorithm A_{opt} adds all ABox assertions (only $Q(a, b)$ here) to the initially empty graph $\mathcal{G}_{\text{ex}_5}$ (Step 1 of Algorithm A_{opt}). (II) In the first iteration of Step 2, Algorithm A_{opt} checks that all of the nodes $A(a)$, $S(b, a)$, $B_1(a)$, $B_2(a)$ and $R(b, a)$ have corresponding SWD paths starting from $Q(a, b)$. Thus, Algorithm A_{opt} adds these nodes to $\mathcal{G}_{\text{ex}_5}$ immediately. We take an example of node $B_1(a)$, for which an SWD path $(Q(a, b), A(a), B_1(a))$ exists. When updating $\mathcal{G}_{\text{ex}_5}$ by adding $B_1(a)$ to it, Algorithm A_{opt} first checks whether the parent node $A(a)$ of $B_1(a)$ has been in $\mathcal{G}_{\text{ex}_5}$; if $A(a)$ is already in $\mathcal{G}_{\text{ex}_5}$, $B_1(a)$ is added to $\mathcal{G}_{\text{ex}_5}$ by creating an edge pointing from $A(a)$ to $B_1(a)$; if $A(a)$ has not been added to $\mathcal{G}_{\text{ex}_5}$, Algorithm A_{opt} adds $A(a)$ ($B_1(a)$, respectively) to $\mathcal{G}_{\text{ex}_5}$ and creates an edge pointing from $Q(a, b)$ to $A(a)$ (an edge pointing from $A(a)$ to $B_1(a)$, respectively).[3] The

[3]These two cases may happen simultaneously, since node $A(a)$ and node $B_1(a)$ are being processed in parallel.

other nodes $A(a)$, $S(b,a)$, $B_2(a)$ and $R(b,a)$ are processed similarly. (III) The two nodes $\bot(a)$ and $\bot(b)$ have no SWD path in the first iteration of Step 2. They are left to be processed in the second iteration. (IV) Finally, Algorithm A_{opt} finishes constructing $\mathcal{G}_{\text{ex}_5}$ after two iterations (Step 3).

From the above example, we can observe that SWD paths exist for all nodes except $\bot(a)$ and $\bot(b)$ in the first iteration of Algorithm A_{opt}. Further, we give the following lemma that holds for any DL-Lite$_{\text{core}}$ or DL-Lite$_{\mathcal{R}}$ ontology.

Lemma 3. *For any DL-Lite$_{core}$ or DL-Lite$_{\mathcal{R}}$ ontology $\mathcal{O}$, there exists a materialization graph $\mathcal{G}$ such that each atom of the form $A(x)$ ($A \neq \bot$) or $R(x,y)$ in $\mathcal{G}$ has an SWD path.*

The above lemma guarantees that all atoms of the form $A(x)$ ($A \neq \bot$) or $R(x,y)$ have to be added to the constructed materialization graph in the first iteration of Step 2 by applying Algorithm A_{opt}. For each atom of the form $\bot(x)$, there may not exist an SWD path in the first iteration of Step 2, since its derivability depends on its two parent nodes (see nodes $\bot(a)$ and $\bot(b)$ in $\mathcal{G}_{\text{ex}_5}$ of Example 5). On the other hand, according to the syntax of DL-Lite, $\bot$ does not occur on the left-hand side of any axiom. In other words, an atom of the form $\bot(x)$ cannot be the parent of any other node. This allows for adding all atoms of the form $\bot(x)$ to the constructed materialization graph in at most two iterations of Step 2 by applying Algorithm A_{opt}. Based on the above discussion, for any DL-Lite ontology $\mathcal{O}$, there always exists a poly-logarithmically bounded function ψ such that Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ can handle the materialization of $\mathcal{O}$; more precisely, we can set that $\psi = 2$. This result is consistent with that by Calvanese et al. [15]. Formally, we use $\mathcal{D}_{dl\_lite}$ to denote the set of all DL-Lite$_{\text{core}}$ and DL-Lite$_{\mathcal{R}}$ ontologies and give the following theorem.

Theorem 2. *There exists a poly-logarithmically bounded function ψ such that $\mathcal{D}_{dl\_lite} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

4.2. Parallel Tractability of DHL Materialization

In this part, we study whether Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ can handle DHL ontologies. Unfortunately there exist DHL ontology classes such that Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ does not work for any poly-logarithmically bounded function ψ. In the following, we first give such a case to illustrate the reason why Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ cannot work. Based on the analysis of this case, we propose to restrict the usage of DHL in order to achieve parallel tractability of materialization.

We find that, an unlimited usage of axioms of the form $B_1 \sqcap B_2 \sqsubseteq A$ makes it impossible for Algorithm A_{opt} to construct a materialization graph in a poly-logarithmic number of iterations of Step 2. We use the following example to illustrate this.

Example 6. *Given a DHL ontology $\mathcal{O}_{ex_6}$ where the TBox contains three axioms: $B_1 \sqcap B_2 \sqsubseteq A$, $\exists S.A \sqsubseteq B_1$ and $\exists R.A \sqsubseteq B_2$; the ABox is $\{S(a_i, a_{i-1}), R(a_i, a_{i-1}), A(a_1)\}$ for $2 \leq i \leq k$ and k an integer greater than 2. We denote the corresponding datalog program of $\mathcal{O}_{ex_6}$ by $P_{ex_6} = \langle R, \boldsymbol{I} \rangle$, where R contains three rules: '$B_1(x), B_2(x) \rightarrow A(x)$', '$S(x,y), A(y) \rightarrow B_1(x)$' and '$R(x,y), A(y) \rightarrow B_2(x)$'. The materialization graph of P_{ex_6} constructed by A_{opt} is denoted by $\mathcal{G}_{ex_6}$ (see Figure 4 (left)).*

One can check that $\mathcal{G}_{\text{ex}_6}$ is the unique materialization graph of P_{ex_6}. Observe that there exists a path (e.g., $A(a_1), B_1(a_2), A(a_2), \ldots, A(a_k)$) between $A(a_1)$ and $A(a_k)$. When performing Algorithm A_{opt} on P_{ex_6}, it can be checked that each node of the form $A(a_i)$ (filled with black color, and $2 \leq i \leq k$) has no SWD path until the i^{th} iteration; in the i^{th} iteration, the parent nodes of $A(a_i)$ have been added to $\mathcal{G}_{\text{ex}_6}$, and, $A(a_i)$ can also be added to $\mathcal{G}_{\text{ex}_6}$. Similar to the datalog

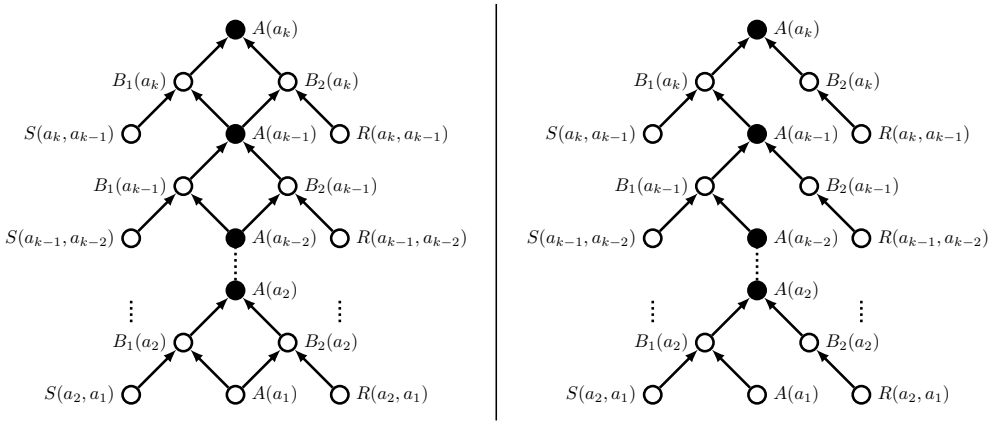

Figure 4: The materialization graph of O_{ex_6} (left) and of O_{ex_7} (right)

program class $\mathbb{P}_{\text{ex}_1}$, we can also get a datalog program class $\mathbb{P}_{\text{ex}_6}$ for the ontology O_{ex_6} when k is a variable. Based on the above analysis, Algorithm A_{opt} cannot complete the materialization for the datalog programs of $\mathbb{P}_{\text{ex}_6}$ in a poly-logarithmic number of iterations. The intuitive reason is that at least two paths exist from $A(a_1)$ to $A(a_k)$. These paths *twist* mutually and share the same joint nodes (see the black nodes). This invalidates the optimization used in Algorithm A_{opt}. That is, for each node $A(a_i)$, $2 \leq i \leq k$, until its parents ($B_1(a_i)$ and $B_2(a_i)$) are added to $\mathcal{G}_{\text{ex}_6}$, there would not exist an available SWD path for $A(a_i)$. We use the term '*path twisting*' to refer to such cases.

In order to make Algorithm A_{opt} terminate in a poly-logarithmic number of iterations, we consider restricting the usage of axioms of the form $B_1 \sqcap B_2 \sqsubseteq A$ to avoid 'path twisting'. An intuitive idea is to ensure that *there is only one path between each two atoms of the form $A(x)$ generated from the rules corresponding to (T1).* We explain it by using the following example where the ontology is modified from that in Example 6.

Example 7. *Consider an ontology O_{ex_7} where the TBox contains three axioms: $B_1 \sqcap B_2 \sqsubseteq A$, $\exists S.A \sqsubseteq B_1$ and $B_3 \sqsubseteq B_2$; the ABox is $\{S(a_i, a_{i-1}), B_3(a_i), A(a_1)\}$ for $2 \leq i \leq k$ and k an integer greater than 2. We denote the corresponding datalog program by P_{ex_7} where the rule set contains: '$B_1(x), B_2(x) \rightarrow A(x)$','$S(x, y), A(y) \rightarrow B_1(x)$' and '$B_3(x) \rightarrow B_2(x)$'. P_{ex_7} has a unique materialization graph denoted by $\mathcal{G}_{ex_7}$ (see Figure 4 (right)).*

In the above example, for the axiom $B_1 \sqcap B_2 \sqsubseteq A$, all derived atoms of the form $B_2(x)$ must not be child nodes of an atom $A(y)$ for some y. This ensures that only one path exists between each two nodes among $A(a_2), \ldots, A(a_k)$. Further, when constructing $\mathcal{G}_{\text{ex}_7}$, Algorithm A_{opt} can terminate after two iterations of Step 2. Specifically, in the first iteration, Algorithm A_{opt} adds all of the nodes $B_3(a_i)$ and $B_2(a_i)$, $2 \leq i \leq k$, to $\mathcal{G}_{\text{ex}_7}$ since they have corresponding SWD paths; after that, all other nodes can be added to $\mathcal{G}_{\text{ex}_7}$ in the second iteration (because each node has an SWD path). Motivated by this example, we consider restricting the usage of the axioms $B_1 \sqcap B_2 \sqsubseteq A$ such that all atoms of the form $B_1(x)$ or $B_2(x)$ cannot be generated by an atom $A(y)$ for some y. To this end, we first define *simple concepts* as follows:

Definition 3. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, a concept $A \in \boldsymbol{CN}$ is simple, if (1) A does not*

occur on the right-hand side of some axiom; or (2) A satisfies the following conditions:

1. *for each $B \sqsubseteq A \in \mathcal{T}$, B is simple;*
2. *for each $\exists R.B \sqsubseteq A \in \mathcal{T}$, B is simple;*
3. *there is no axiom of the form $B_1 \sqcap B_2 \sqsubseteq A$ in $\mathcal{T}$.*

Based on simple concepts, we restrict DHL ontologies such that, in all axioms of the form $B_1 \sqcap B_2 \sqsubseteq A$, at least one concept of B_1 and B_2 should be a simple concept (we call this the *simple-concept restriction*). Intuitively, for such restricted DHL ontologies, the situation of 'path twisting' does not happen. This is because, for each axiom of the form $B_1 \sqcap B_2 \sqsubseteq A$ such that, w.l.o.g., B_1 is a simple concept, none of the ancestors of $B_1(x)$ for some x is generated from the rules corresponding to (T1).

Example 8. *In the ontology of Example 6, all of A, B_1 and B_2 are non-simple concepts. In the ontology of Example 7, A and B_1 are non-simple concepts, while B_3 and B_2 are simple concepts. Further, it can be checked that the ontology of Example 7 follows the simple-concept restriction and can be handled by Algorithm A^{ψ}_{opt} for some poly-logarithmic function ψ.*

We define the following class of DHL ontologies based on the above restriction and give Theorem 3 to show that any DHL ontology that satisfies the simple-concept restriction can be handled by Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ for some poly-logarithmic function ψ.

Definition 4. *Let $\mathcal{D}_{dhl}$ be a class of datalog programs where each program is rewritten from a DHL ontology that follows the condition that, for all axioms of the form $A_1 \sqcap A_2 \sqsubseteq B$, at least one concept of A_1 and A_2 should be a* simple concept.

Theorem 3. *There exists a poly-logarithmically bounded function ψ s.t. $\mathcal{D}_{dhl} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

4.3. Parallel Tractability of DHL(∘) Materialization

In this part, we study materialization tractable in parallel for DHL(∘) ontologies. In addition to the rules in DHL, we also have to consider complex RIAs (R4). We next show that complex RIAs may also cause the situation of 'path twisting'. Consider the following example:

Example 9. *Given a DHL(∘) ontology O_{ex_9} where the TBox is empty; the RBox $\mathcal{R}$ contains three axioms: $R_1 \circ R_2 \sqsubseteq R$, $R_3 \circ R \sqsubseteq R_1$ and $R \circ R_4 \sqsubseteq R_2$; the ABox $\mathcal{A}$ is $\{R(a_1, a_1), R_3(a_i, a_{i-1}), R_4(a_{i-1}, a_i)\}$ for $2 \leq i \leq k$ and k an integer greater than 2. The corresponding datalog program P_{ex_9} contains three rules: '$R_1(x, y), R_2(y, z) \rightarrow R(x, z)$', '$R_3(x, y), R(y, z) \rightarrow R_1(x, z)$' and '$R(x, y), R_4(y, z) \rightarrow R_2(x, z)$'. The materialization graph of P_{ex_9} constructed by Algorithm A_{opt} is denoted by $\mathcal{G}_{ex_9}$.*

One can check that the materialization graph $\mathcal{G}_{\text{ex}_9}$ has the same shape as that of $\mathcal{G}_{\text{ex}_6}$ in Figure 4. A twisted path exists in $\mathcal{G}_{\text{ex}_9}$ involving $R(a_i, a_i)$, $2 \leq i \leq k$, as the joint nodes. Further, all the roles R_1, R_2, R_3, R_4 and R in this example are non-transitive roles.

Inspired by what we do for axioms $B_1 \sqcap B_2 \sqsubseteq A$, we require that, for all axioms of the form $R_1 \circ R_2 \sqsubseteq R$, if R is not a transitive role and no transitive role S exists such that $R \sqsubseteq_* S$, then, at least one of R_1 and R_2 is a *simple role*.[4] We now consider such an axiom $R_1 \circ R_2 \sqsubseteq R$

[4] See the definition of a simple role in Section 2.

(denoted by α_1) where R is a transitive role. That is, we also have $R \circ R \sqsubseteq R$ (denoted by α_2). By replacing R on the left-hand side of α_2 using R_1 and R_2, we can get a complex RIA in the form of $R_1 \circ R_2 \circ R_1 \circ R_2 \sqsubseteq R$ (denoted by α_3). If one of R_1 and R_2 is not a simple role, the corresponding rule of α_3 may also lead to 'path twisting'.[5] This can be explained as follows: Without loss of the generality, R_2 is a simple role, while R_1 is not. For some atom $R(x, y)$, it may depend on two different nodes of the predicate R_1 through the corresponding rule of α_3. A similar analysis applies to the cases of α_1 where R is not a transitive role, while another transitive role S exists such that $R \sqsubseteq_* S$. That is, we can obtain a complex RIA of the form $R_1 \circ R_2 \circ R_1 \circ R_2 \sqsubseteq S$. Further, the situation of path twisting also exists. To tackle the above issue, we require both of R_1 and R_2 in α_1 to be simple roles (we call the above restriction for transitive and non-transitive roles the *simple-role restriction*). Combined with the simple-concept restriction, we define a class of DHL(∘) ontologies as follows:

Definition 5. *$\mathcal{D}_{dhl(\circ)}$ is a class of datalog programs where each program is rewritten from a DHL(∘) ontology and the following conditions are satisfied:*

1. *for all axioms of the form $A_1 \sqcap A_2 \sqsubseteq B$, at least one concept of A_1 and A_2 is a* simple concept*;*
2. *for all axioms of the form $R_1 \circ R_2 \sqsubseteq R$, if there exits a transitive role S such that $R \sqsubseteq_* S$, then both R_1 and R_2 are* simple roles*; otherwise at least one of R_1 and R_2 is a* simple role.

Example 10. *For the ontology O_{ex_9} in Example 9, all of the roles R_1, R_2 and R are non-simple roles. Thus, O_{ex_9} does not follow the simple-role restriction because of $R_1 \circ R_2 \sqsubseteq R$. Consider the ontology O_{ex_1} in Example 1 again. The role R is a non-simple role, while S is a simple role. Thus, O_{ex_1} follows the simple-role restriction. All the implicit nodes in $\mathcal{G}_{ex_1}$ have corresponding SWD paths in the first iteration. Thus, 'path twisting' cannot occur when materializing O_{ex_1} by Algorithm A_{opt}.*

We further give Theorem 4 to show that Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$ can handle all datalog programs in $\mathcal{D}_{dhl(\circ)}$ for some poly-logarithmic function ψ.

Theorem 4. *There exists a poly-logarithmically bounded function ψ s.t. $\mathcal{D}_{dhl(\circ)} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

4.4. Parallel Tractability of Reasoning over RDFS Ontologies

In this part, we discuss parallel tractability of reasoning over RDFS ontologies. Although RDFS is not directly based on a description logic, it can (partly) be described by a set of DL axioms [16]. The correspondence between RDFS statements and DL axioms is listed in Table 2. We call RDFS statements of the form (S1–S4) *the schema data* (which correspond to TBox axioms), statements of the form (S5) and (S6) *the instance data* (which correspond to ABox axioms).

The original rule set for RDFS reasoning is given by Hayes [25]. By applying the rules in this rule set, new schema data could be derived. Thus, the reasoning task of RDFS is different from the task of materialization. Further, an RDFS ontology allows statements about blank nodes which act like variables. This possibly leads to infinite entailments. Additionally, the original rule set is incomplete with the RDF restriction that blank nodes cannot occur in predicate positions [20], however, by allowing blank nodes in predicate position the rule set turns out to be complete.

[5] Obviously, applying the rules of α_1 and α_2 separately has the same effect to that of only applying the rule of α_3.

Table 2: RDFS statements and the corresponding DL axioms

	RDFS statements	Axiom
(S1)	P `rdfs:domain` C	$\top \sqsubseteq \forall P^{-}.C$
(S2)	P `rdfs:range` C	$\top \sqsubseteq \forall P.C$
(S3)	B `rdfs:subClassOf` C	$B \sqsubseteq C$
(S4)	R `rdfs:subPropertyOf` S	$R \sqsubseteq S$
(S5)	a `rdf:type` C	$C(a)$
(S6)	a R b	$R(a, b)$

The computational complexity of RDFS reasoning is NC-complete [20], which is not considered to be tractable in parallel.

The situation of path twisting may also happen when conducting reasoning on RDFS ontologies. Suppose R_{subp} denotes the RDFS built-in property `rdfs:subPropertyOf`. We define three further properties R_1, R_2 and R_3 by the following axioms: $R_1 \sqsubseteq R_{\text{subp}}$, $R_2 \sqsubseteq R_{\text{subp}}$ and $R_{\text{subp}} \sqsubseteq R_3$. It can be checked that these three axioms entail the complex RIA $R_1 \circ R_2 \sqsubseteq R_3$ (denoted by β). If axiom β does not meet the simple-role restriction (see Section 4.3), the situation of path twisting may also happen as shown in Example 9.

In this work, we study the task of materialization, hence, we can only focus on newly entailed instance data for the RDFS ontologies. To this end, we assume that any class of RDFS ontologies has the same schema data (I). In this way, axioms such as β cannot contribute to the computational complexity, since they only apply to schema data. Further, we assume blank nodes not to occur in any class of RDFS ontologies (II). This allows for expressing RDFS ontologies in DHL (see Table 2). Based on the two restrictions (I) and (II), the materialization of RDFS ontologies is tractable in parallel (see Section 4.2).

5. Further Optimizing DHL(∘) Materialization

In this section, we first discuss why Algorithm A_{opt} can hardly work in practice. In order to make Algorithm A_{opt} more practical, we propose to modify Algorithm A_{opt} and give an algorithm variant Algorithm A_{prc}. We show that Algorithm A_{prc} can also be restricted to an NC version when materializing DHL(∘) ontologies that satisfy both the simple-concept and the simple-role restriction.

5.1. Reducing Computing Space

In previous sections, Algorithm A_{opt} is mainly used for theoretical analysis. However, this algorithm can hardly work in practice due to its inherently high requirement of computing space. Specifically, Algorithm A_{opt} constructs a materialization graph of the given datalog program P by checking all possible rule instantiations in P^*. One can check that $|P^*|$ could be the quadratic or cubic in the number of constants occurring in P. Recall the analysis of the parallel complexity of Algorithm A_{bsc} and Algorithm A_{opt}. The number of constants is at most $|\mathbf{I}|^e$ where e denotes the maximum arity of any predicate in $\mathbf{I}$. Consider a datalog rule of the form '$R(x, y), A(y) \rightarrow B(x)$'. Since there are two variables in this rule, the number of all possible rule instantiations is $(|\mathbf{I}|^e)^2$. Similarly, for a datalog rule rewritten from an axiom of the form (R3) or (R4), the number of all possible rule instantiations is $(|\mathbf{I}|^e)^3$. Undoubtedly, a plain implementation of Algorithm A_{opt} would be slow when the target datalog program or ontology tends to be large in size. On the

other hand, from Examples 1, 5 and 6, we can observe that the rule instantiations used for the construction of materialization graphs always cover a small part of P^* with respect to the target datalog program P. Thus, we consider reducing the computing space of Algorithm A_{opt} by narrowing down the scope of rule instantiations to be checked. Our strategy is to restrict that, *in each iteration of Algorithm A_{opt}, each of the checked rule instantiations should involve at least one body atom that has been added to the constructed materialization graph*. Since DHL($\circ$) is our focus, we explain how to apply the above strategy in DHL($\circ$) materialization as follows.

We first consider a datalog rule of the form '$A(x) \rightarrow B(x)$' that corresponds to (T1). The above strategy requires that, in each iteration, Algorithm A_{opt} can only check the rule instantiations of the form '$A(a) \rightarrow B(a)$' where $A(a)$ has been added to the constructed materialization graph $\mathcal{G}$. If there are n atoms of the form $A(x)$ that have been added to $\mathcal{G}$, the number of all checked rule instantiations of the above rule is also n, instead of $(|\mathbf{I}|e)$. This is because, for each assertion $A(a)$, the datalog rule '$A(x) \rightarrow B(x)$' has only one rule instantiation '$A(a) \rightarrow B(a)$' where the variable x is substituted by the constant a. The cases of (R1) and (R2) can be analyzed similarly. We now consider datalog rules that have two body atoms, i.e., datalog rules of the form '$R(x, y), A(y) \rightarrow B(x)$' (see (T3)), '$A_1(x), A_2(x) \rightarrow B(x)$' (see (T2)) and '$R_1(x, y), R_2(y, z) \rightarrow R(x, z)$' (see (R3) and (R4)). We require that, for each rule instantiation of these rules checked by Algorithm A_{opt}, at least one body atom has been added to the constructed materialization graph $\mathcal{G}$; thus, it can be checked that, in each iteration of Algorithm A_{opt}, the number of checked rule instantiations is at most $k(|\mathbf{I}|e)$ where k is the number of atoms that have been added to $\mathcal{G}$. Note that, for rule instantiations of two body atoms, we do not require that both of these two body atoms have been added to the constructed materialization graph. Otherwise, the algorithm would perform as Algorithm A_{bsc} and, thus, the optimizations used in Algorithm A_{opt} cannot further work.

5.2. Further Optimizing Algorithm A_{opt}

We use the above method of narrowing down the scope of rule instantiations to modify Algorithm A_{opt} and obtain an algorithm variant Algorithm A_{prc}. Recall that, in each iteration of Step 2, Algorithm A_{opt} checks all possible rule instantiations and computes an `rch` relation and its transitive closure to determine the existence of SWD paths. We let Algorithm A_{prc} conduct the same work with narrowing down the scope of rule instantiations to be checked as well. This can be described by the following algorithm.

`Algorithm` PRC. This algorithm has as inputs (1) a DHL($\circ$) ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and its datalog program $P = \langle R, \mathbf{I} \rangle$; (2) a (partial) materialization graph $\mathcal{G} = \langle V, E \rangle$ that is constructed from P. This algorithm outputs an `rch` relation S_{rch} that is computed as follows:

- add $\texttt{rch}(A(a), B(a))$ to S_{rch} where $A(a) \rightarrow B(a) \in P^*$, $\mathcal{O} \models A \sqsubseteq B$ and $A(a) \in V$;
- add $\texttt{rch}(A(b), B(a))$ to S_{rch} where $R(a, b), A(b) \rightarrow B(a) \in P^*$, $\exists R.A \sqsubseteq B \in \mathcal{T}$ and $R(a, b) \in V$;
- add $\texttt{rch}(A_2(a), B(a))$ to S_{rch} where $A_1(a), A_2(a) \rightarrow B(a) \in P^*$, $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{T}$ and $A_1(a) \in V$;
- add $\texttt{rch}(A_1(a), B(a))$ to S_{rch} where $A_1(a), A_2(a) \rightarrow B(a) \in P^*$, $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{T}$ and $A_2(a) \in V$;
- add $\texttt{rch}(R(a, b), S(a, b))$ to S_{rch} where $R(a, b) \rightarrow S(a, b) \in P^*$, $\mathcal{O} \models R \sqsubseteq S$ and $R(a, b) \in V$;

- add $\texttt{rch}(R(a,b), S(b,a))$ to S_{rch} where $R(a,b) \rightarrow S(b,a) \in P^*$, $\mathcal{O} \models R \sqsubseteq S^-$ and $R(a,b) \in V$;
- add $\texttt{rch}(R_2(b,c), R_3(a,c))$ to S_{rch} where $R_1(a,b), R_2(b,c) \rightarrow R_3(a,c) \in P^*$, $R_1 \circ R_2 \sqsubseteq R_3 \in \mathcal{R}$ (the case where $R_1 \equiv R_2 \equiv R_3$ is also included) and $R_1(a,b) \in V$;
- add $\texttt{rch}(R_1(a,b), R_3(a,c))$ to S_{rch} where $R_1(a,b), R_2(b,c) \rightarrow R_3(a,c) \in P^*$, $R_1 \circ R_2 \sqsubseteq R_3 \in \mathcal{R}$ and $R_2(b,c) \in V$. □

Based on Algorithm PRC, we modify Algorithm A_{opt} by replacing Step (**i**) of Algorithm OPT with Algorithm PRC. Algorithm A_{prc} is, thus, given as follows:

`Algorithm` $\mathsf{A}_{\texttt{prc}}$. Given a DHL(∘) ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and its datalog program $P = \langle R, \mathbf{I} \rangle$, this algorithm returns a materialization graph $\mathcal{G}$ of P. Initially $\mathcal{G}$ is empty. The following steps are then performed:

(*Step 1*) Add all facts in $\mathbf{I}$ to $\mathcal{G}$.

(*Step 2*) Compute S_{rch} by performing Algorithm PRC; use an `NC` algorithm to compute the transitive closure S^*_{rch} (see (**ii**) in Algorithm OPT); update $\mathcal{G}$ by performing (**iii**) in Algorithm OPT.

(*Step 3*) If no node has been added to $\mathcal{G}$ (in Step 2), terminate, otherwise iterate Step 2. □

Theorem 5. *For any DHL(∘) ontology $\mathcal{O}$, Algorithm A_{prc} halts and outputs a materialization graph of $\mathcal{O}$.*

The above theorem is given to show the correctness of Algorithm A_{prc}. Since Algorithm A_{prc} is almost identical with Algorithm A_{opt}, it has the same upper bound of the computing time, i.e., $c_1 + \ell' c_3 + \ell' c_4 \log(|\mathbf{I}|e) + \ell' c_5 \log^2(|\mathbf{I}|e)$, where ℓ' denotes the number of iterations of Step 2, and the symbols c_1, c_3, c_4, c_5 have the same meanings as those in the analysis of Algorithm A_{opt}. The difference between the two algorithms is that Algorithm A_{prc} considers a smaller scope of rule instantiations. This gives rise to smaller sizes of the relation S_{rch}. Based on the above discussion and Algorithm PRC, the size of relation S_{rch} is bounded by $h(|\mathbf{I}|e)$ (where h denotes the number of nodes that have been added to the graph), while the size of the relation S_{rch} as computed by Algorithm A_{opt} is at most $p^2(|\mathbf{I}|e)^{2w}$.

We next use an example to show how Algorithm A_{prc} handles DHL(∘) materialization.

Example 11. *Consider performing Algorithm A_{prc} on the ontology $\mathcal{O}_{ex_1}$ of Example 1. Note that the individual set **IN** is $\{a_1, ..., a_k, b\}$, i.e., $k+1$ individuals are involved. Initially, Algorithm A_{prc} adds all the facts $(A(b), R(a_1,b), S(a_2,a_1), ..., S(a_k,a_{k-1}))$ to the result $\mathcal{G}_{ex_1}$ (Step 1). In the first iteration of Step 2, Algorithm A_{prc} computes S_{rch} first. According to Algorithm PRC, for each atom of the form $S(a_i,a_{i-1})$, $2 \leq i \leq k$, and $\forall o \in$ **IN**, an `rch` relation of the form $\texttt{rch}(R(a_{i-1},o), R(a_i,o))$ is added to S_{rch}. Since the atom $R(a_1,b)$ has been added to $\mathcal{G}_{ex_1}$, Algorithm A_{prc} checks that all atoms of the form $R(a_i,b)$, $2 \leq i \leq k$, have SWD paths and are added to $\mathcal{G}_{ex_1}$ in the first iteration. In the second iteration of Step 2, since all atoms of the form $R(a_i,b)$, $1 \leq i \leq k$, have been in $\mathcal{G}_{ex_1}$, the `rch` relations of the form $\texttt{rch}(A(b), A(a_i))$ are checked with respect to the axiom $\exists R.A \sqsubseteq A$; further all atoms of the form $A(a_i)$, $1 \leq i \leq k$, are finally added to $\mathcal{G}_{ex_1}$ by Algorithm A_{prc}.*

From the above example, one can find that Algorithm A_{prc} terminates after two iterations of Step 2. This is the same as Algorithm A_{opt} (see Example 4). We use Theorem 6 to show that Algorithm A_{prc} also has an NC version when handling ontologies that satisfy the simple-concept and the simple-role restrictions. The correctness of this theorem is based on the fact that the method of narrowing down the scope of rule instantiations in Algorithm A_{prc} does not influence the determination of the existence of SWD paths. A detailed analysis can be found in the proof in the appendix.

Theorem 6. *For any DHL($\circ$) ontology $\mathcal{O}$ that satisfies the simple-concept and the simple-role restriction, there exists a poly-logarithmically bounded function ψ, such that Algorithm A_{prc}^{ψ} outputs a materialization graph of $\mathcal{O}$.*

6. Evaluation and Analysis

In the first part of this section, we analyze the well-known benchmark LUBM and the real-world dataset YAGO and show cases where these ontologies belong to the parallel tractability classes. In the second part, we evaluate the implementation of Algorithm A_{prc} and compare it to the reasoning system RDFox. In the third part, we examine the effects of the depths of materialization graphs based on two modified versions of LUBM.

6.1. Parallel Tractability of LUBM Datasets and YAGO

LUBM. In the Semantic Web community, the Lehigh University Benchmark (LUBM) is proposed to facilitate the evaluation of ontology-based systems in a standard and systematic way. In the latest version of LUBM,[6] the core ontology contains 48 classes and 32 properties, used to describe the departments and the staff of universities. By setting a different number of universities for an ontology-generator, users can get datasets of any size based on the core ontology. For the simple form of the core ontology, the statements about properties, such as inverse property statements, can be rewritten into datalog rules of the form (R1), (R2) and (R3) in Table 1. Most of the statements about classes can be rewritten into datalog rules of the form (T1) and (T3) in Table 1. There are six axioms of form (T2) as listed below:

$$\texttt{Person} \sqcap \texttt{CourseTaker} \sqsubseteq \texttt{Student} \tag{α_1}$$
$$\texttt{Person} \sqcap \texttt{OrganizationWorker} \sqsubseteq \texttt{Employee} \tag{α_2}$$
$$\texttt{Person} \sqcap \texttt{DepartmentHead} \sqsubseteq \texttt{Chair} \tag{α_3}$$
$$\texttt{Person} \sqcap \texttt{ProgramHead} \sqsubseteq \texttt{Director} \tag{α_4}$$
$$\texttt{Person} \sqcap \texttt{CourseAssistant} \sqsubseteq \texttt{TeachingAssistant} \tag{α_5}$$
$$\texttt{Person} \sqcap \texttt{CollegeHead} \sqsubseteq \texttt{Dean} \tag{α_6}$$

In the above axioms, the six concepts CourseTaker, OrganizationWorker, DepartmentHead, ProgramHead, CourseAssistant and CollegeHead have corresponding definitions. For example, the concept CourseTaker is defined by:

$$\texttt{CourseTaker} \equiv \exists \texttt{take.Course},$$

[6] http://swat.cse.lehigh.edu/projects/lubm/

which is equivalent to the two axioms of simple form,

$$\exists \texttt{take.Course} \sqsubseteq \texttt{CourseTaker} \text{ and} \tag{1}$$

$$\texttt{CourseTaker} \sqsubseteq \exists \texttt{take.Course}. \tag{2}$$

Axiom (1) corresponds to form (T3). Axiom (2) requires existentially quantified variables in the rule head when rewriting the axiom into a logic rule:

$$\texttt{CourseTaker}(x) \rightarrow \exists y(\texttt{take}(x, y) \wedge \texttt{Course}(y)) \tag{τ}$$

where a free variable y is introduced. This kind of axiom is a general case of (T4) in Table 1 for which A is actually replaced by the top concept $\top$. Similarly to how we handle (T4), we can also eliminate the free variable y in Rule (τ) via Skolemization, i.e., by replacing the variable y with a new constant o. In this way, Rule (τ) can be rewritten into $\texttt{CourseTaker}(x) \rightarrow \texttt{take}(x, o) \wedge \texttt{Course}(o)$. If we only focus on the materialization task, the rewriting approach via Skolemization guarantees correctness [26]. On the other hand, Rule (τ) is not considered when using OWL RL reasoners to handle LUBM [11, 27]. In summary, if the rewriting approach is used for the above kind of rule, the core ontology can be expressed in DHL. We can further check that the concepts occurring in (α_1–α_6) are all simple concepts. Thus, the materialization of LUBM datasets is tractable in parallel and can be handled by Algorithm A_{prc}.

YAGO. The knowledge base YAGO[7] is constructed from Wikipedia and WordNet. The latest version YAGO3 [28] has millions of facts. In order to balance the expressiveness and computing efficiency, a YAGO-style language, called the *YAGO model*, is proposed based on a slight extension of RDFS [29]. The YAGO model defines a set of properties: `domain, range, subClassOf, subRelationOf` and `type`, and a set of classes: `entity, class, relation` and `acyclicTransitiveRelation`. The facts in the YAGO model are stated by triples, e.g., $(r_1, \texttt{subRelationOf}, r_2)$, which are similar to RDFS statements. A group of rules for reasoning over YAGO ontologies is specified by Suchanek et al. [29]:

$$(r, \texttt{domain}, c), (x, r, y) \rightarrow (x, \texttt{type}, c) \tag{3}$$

$$(r, \texttt{range}, c), (x, r, y) \rightarrow (y, \texttt{type}, c) \tag{4}$$

$$(c_1, \texttt{subClassOf}, c_2), (x, \texttt{type}, c_1) \rightarrow (x, \texttt{type}, c_2) \tag{5}$$

$$(r_1, \texttt{subRelationOf}, r_2), (x, r_1, y) \rightarrow (x, r_2, y) \tag{6}$$

$$(r, \texttt{type}, \texttt{acyclicTransitiveRelation}), (x, r, y), (y, r, z) \rightarrow (x, r, z) \tag{7}$$

According to the semantics given to YAGO [29], the built-in properties in YAGO, i.e., `domain, range, subClassOf, subRelationOf` and `type` act in the same way as the terms in RDFS statements of the form (S1–S5) in Table 2, respectively. In contrast to RDFS, YAGO also allows for defining transitive properties using the class `acyclicTransitiveRelation`; further, any fact in some YAGO ontology cannot be described using blank nodes. By carefully checking the rules (3–7) for reasoning over YAGO ontologies, one can see that these rules can be rewritten into datalog rules of the form (T3),[8] (T1), (R1) and (R3) in Table 1, respectively. Based on the above analysis, any YAGO ontology can be expressed in DHL and satisfies the simple-concept and the

[7] http://www.mpi-inf.mpg.de/home/

[8] Both of Rule (3) and Rule (4) can be rewritten into (T3).

Table 3: The statistics of the test ontologies, where ♯concept denotes the number of concept names, ♯role the number of role names, ♯axiom the number of TBox and RBox axioms, ♯individual the number of individuals and ♯assertion the number of assertions in the ontology

ontology	♯concept	♯role	♯axiom	♯individual	♯assertion
lubm-50				1,082,818	11,601,923
lubm-100				2,179,766	23,837,579
lubm-150	48	32	99	3,243,523	35,466,709
lubm-200				4,341,309	46,537,764
lubm-250				5,421,894	58,125,155
yago-core	65,318	74	55,615	4,077,882	45,277,896

Table 4: The reasoning time results in seconds

	♯thread	lubm-50	lubm-100	lubm-150	lubm-200	lubm-250	yago-core
ParallelDHL	1	93.66	213.75	376.39	498.34	693.49	773.06
	4	21.71	54.03	111.2	123.76	159.82	223.4
	8	9.44	23.04	55.78	67.29	86.41	98.33
	16	4.02	12.7	20.9	35.63	43.91	47.08
	24	4.06	9.82	16.84	22.31	27.47	34.48
RDFox	1	23.01	42.8	71.92	88.64	111.96	105.73
	4	5.97	11.64	18.5	22.31	25.59	35.5
	8	4.88	5.89	9.95	11.21	12.8	19.07
	16	3.18	3.75	8.37	9.16	12.19	13.22
	24	1.92	3.9	6.21	7.74	9.58	11.81

simple-role restrictions. We then have that, for any well-constructed class of YAGO ontologies, Algorithm A_{prc} can handle all of the ontologies in the class.

In addition to LUBM and YAGO, we further investigate different kinds of ontologies and datasets including benchmarks, real-world ontologies and datasets that can be expressed in ontology languages. These ontologies and datasets are collected from the Protégé ontology library,[9] Swoogle[10] and the Oxford ontology library.[11] Based on the analysis of these ontologies, we found that, ignoring imports, many of them belong to $\mathcal{D}_{dhl}$ or $\mathcal{D}_{dhl(\circ)}$. All of these investigated ontologies and the analysis results are available online.[12]

6.2. Evaluating the Implementation of Algorithm A_{prc}

We implement a prototype system ParallelDHL for DHL(∘) materialization based on Algorithm A_{prc}. Since ParallelDHL is evaluated on a platform which has limited memory space and a fixed number of processors, the parallel assumption given in Section 3.3 does not apply to ParallelDHL. In the implementation of ParallelDHL, we use a hash function to map any rule instance to the identifier of some processor. Thus, when performing ParallelDHL, each processor handles a group of rule instances.

[9] http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library
[10] http://swoogle.umbc.edu/
[11] http://www.cs.ox.ac.uk/isg/ontologies/lib/
[12] https://github.com/quanzz/PT

We run ParallelDHL on LUBM and YAGO to further analyze these two kinds of ontologies. We also use the reasoning system RDFox [4] for comparison. We use five generated LUBM datasets, lubm-50, lubm-100, lubm-150, lubm-200 and lubm-250, where lubm-50 contains the descriptions of 50 universities (and analogously for the other datasets). For YAGO, we use its core version, denoted by yago-core.[13] The yago-core ontology is a subset of the full YAGO dataset without any imports. It does not include datatype statements, the links between different data sources, the degrees of confidence and other kinds of annotations that we do not consider in this work. The statistics of the above ontologies is given in Table 3. The running platform for this experiment is a server with 50 Gigabyte RAM and eight physical cores, in each core three logical threads can be allocated.

We materialize the LUBM datasets and the yago-core ontology with ParallelDHL and RDFox using different numbers of threads (1, 4, 8, 16 and 24, respectively). The reasoning times are given in Table 4. We also give Figure 5 to graphically compare the two systems. In Figure 5(left), the abscissa records the five LUBM datasets and the ordinate records the reasoning times in seconds with 24 threads being allocated. In Figure 5(right), the abscissa records the numbers of threads and the ordinate records the reasoning times in seconds over the yago-core ontology.

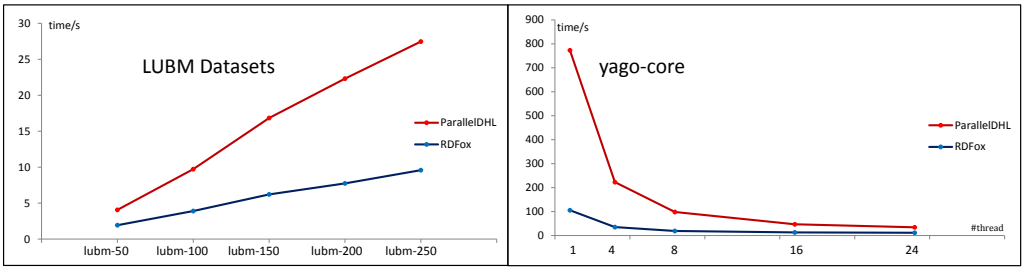

Figure 5: (left) The reasoning times over the LUBM ontologies; (right) the reasoning times over yago-core

From Table 4, we can see that for lubm-50 the two systems perform similarly. For the other ontologies, ParallelDHL scales comparable to RDFox with several threads being allocated. For lubm-250 and yago-core, when only one thread is used, ParallelDHL is significantly slower. The reason is that the computation of the relation S_{rch} requires a large amount of time. When four and more threads are allocated, ParallelDHL has a better efficiency. Although ParallelDHL is by far not as optimized as RDFox, it also shows the scalability. For both of RDFox and ParallelDHL, we can see a linear trend of the reasoning times in Figure 5(left) and an obvious downtrend of the reasoning times with an increasing number of threads in Figure 5(right). This also indicates that ParallelDHL will finish the materialization tasks on the test ontologies in a shorter period of time with more threads being allocated.

6.3. Depths of Materialization Graphs

From the complexity analysis for Algorithm A_{bsc} (see Section 3.3), we have that the computing time of Algorithm A_{bsc} depends on the sizes of the input ontologies and the numbers

[13]The yago-core ontology is available at https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads/.

Table 5: The statistics of the generated datasets

ontology	♯concept	♯role	♯axiom	♯individual	♯assertion	MG-depth
ptu-4*m*-1	51	34	102	4,000,440	8,000,839	8,000,799
ptu-4*m*-2					8,000,837	4,000,399
ptu-4*m*-4					8,000,835	2,000,199
ptu-4*m*-8					8,000,833	1,000,099
ptu-4*m*-16					8,000,831	500,049
nptu-4*m*-1					8,000,839	8,000,799
nptu-4*m*-2					8,000,837	4,000,399
nptu-4*m*-4					8,000,835	2,000,199
nptu-4*m*-8					8,000,833	1,000,099
nptu-4*m*-16					8,000,831	500,049

of iterations of Step 2, which is equal to the *depth of the target materialization graph*, which we abbreviate with *MG-depth* in the remainder. Thus, the MG-depth also determines the computing time of Algorithm A_{bsc} in theory. This conclusion also applies to Algorithm A_{opt} and Algorithm A_{prc} based on the analysis in Section 3.4 and Section 5, respectively. In this part, we examine the effects of MG-depths.

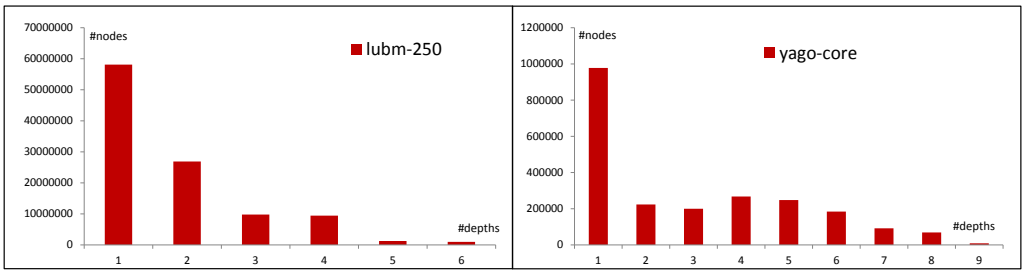

Figure 6: The numbers of nodes in different depths of lubm-250 (left) and yago-core (right)

MG-Depths of LUBM and YAGO. For the five generated LUBM datasets and the yago-core ontology, we record the MG-depth for each ontology and the number of nodes in different depths. We have that all of the LUBM datasets have an MG-depth of 6, and the MG-depth of the yago-core ontology is 9. The two histograms in Figure 6 visualize the number of nodes in different depths for lubm-250 and yago-core, respectively. The abscissa records the depths from the bottom of a materialization graph to the top with the histograms showing the amount of nodes for each depth. From the above results, we can see that the MG-depths are far less than the sizes of the input ontologies. In these cases, the input ontology size turns out to be the dominant factor for the reasoning efficiency. This is also shown in Figure 5(left) for the LUBM datasets, where the reasoning times grow proportionally with the increasing number of universities. Thus, the above experiments hardly show the effects of MG-depths.

Generating Ontologies of Different MG-Depths. In order to examine the effects of MG-depths, we consider modifying the core LUBM ontology such that ontologies of different MG-depths can be obtained. Inspired by Example 7, we add to the core LUBM ontology the following

three axioms to describe the reference relationships among articles:

$$\exists \texttt{referTo}.\texttt{CollegeArticle} \sqsubseteq \texttt{CollegeConference} \qquad (\beta_1)$$

$$\exists \texttt{cite}.\top \sqsubseteq \texttt{CollegeSession} \qquad (\beta_2)$$

$$\texttt{CollegeConference} \sqcap \texttt{CollegeSession} \sqsubseteq \texttt{CollegeArticle} \qquad (\beta_3)$$

where Axiom (β_1) states that any article referring to a college article is published in a college conference, Axiom (β_2) defines that any article having citations is published in a college session and Axiom (β_3) states that any article published in both a college conference and a college session is a college article.

We name the above new ontology PTU. Based on the newly added axioms in PTU, we further modify the ontology-generator of LUBM such that a *reference chain* of articles can be generated as follows: $\texttt{referTo}(\texttt{a}_i, \texttt{a}_{i+1})$ and $\texttt{cite}(\texttt{a}_i, \texttt{a}_{i+1})$ ($i \in \{0, 1, 2, ...\}$), where $\texttt{a}_i$ denotes an article instance. As discussed in Section 6.1, the original core ontology of LUBM is tractable in parallel via Skolemization. Further, it can be checked that the concepts in PTU are all simple concepts. Thus, PTU belongs to the parallel tractability class. For comparison, we create another core ontology, named NPTU, which is not tractable in parallel. NPTU is almost the same as PTU except that Axiom (β_2) in PTU is replaced by:

$$\exists \texttt{cite}.\texttt{CollegeArticle} \sqsubseteq \texttt{CollegeSession} \qquad (\beta_4)$$

which describes that any article citing a college article is published in a college session. It can be checked that NPTU does not satisfy the simple-concept restriction following analogous arguments to those given in Example 6.

The two core ontologies, PTU and NPTU, and the modified ontology-generator are available online.[14] To reduce the effects of ontology sizes, we generate five datasets of the similar sizes for PTU (NPTU, respectively), denoted by ptu-4*m*-*i* (nptu-4*m*-*i*, respectively), $i \in \{1, 2, 4, 8, 16\}$, where i is the number of article reference chains and 4*m* denotes that 4 million articles are involved on average in all of the reference chains. The statistics for the generated datasets are given in Table 5. Table 5 shows that the datasets ptu-4*m*-*i* ($i \in \{1, 2, 4, 8, 16\}$) have an almost identical number of assertions while the MG-depth decreases proportionally, e.g., the MG-depth of ptu-4*m*-1 is $8,000,799$, while the MG-depth of ptu-4*m*-16 is just $500,049$. This is analogous for the datasets nptu-4*m*-*i* ($i \in \{1, 2, 4, 8, 16\}$).

Experimental Results and Analysis. Table 6 compares the reasoning times and speedups[15] of ParallelDHL and RDFox for materializing ptu-4*m*-1 and nptu-4*m*-1.[16] Figure 7 and 8 show the trends of reasoning times. Specifically, the two line graphs of Figure 7 show the reasoning times in seconds of ParallelDHL and RDFox over all the generated datasets with 24 threads being allocated. In Figure 8, the abscissas of the two line graphs record the numbers of threads and the ordinates record the reasoning times over ptu-4*m*-1 and nptu-4*m*-1, respectively.

Figure 7 shows that MG-depths indeed determines the reasoning times. For the five datasets ptu-4*m*-*i*, $i \in \{1, 2, 4, 8, 16\}$, the experimental results show an obvious downtrend of the reasoning times with declining MG-depths for both ParallelDHL and RDFox (see Figure 7(left)). The

[14] https://github.com/quanzz/PT

[15] The speedup is computed by $\frac{T_1}{T_n}$ where T_1 is the computing time using 1 thread and T_n is the computing time using n thread(s).

[16] The detailed experimental results are available online at https://github.com/quanzz/PT.

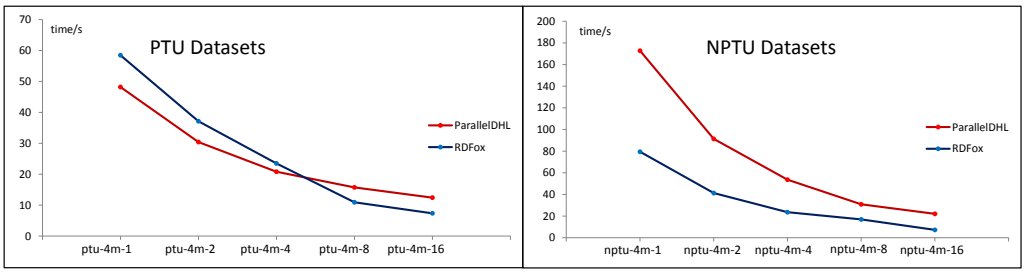

Figure 7: (left) reasoning times over the PTU datasets; (right) reasoning times over the NPTU datasets

Table 6: The reasoning-times in seconds and speedups

	♯thread	ptu-4m-1	speedup	nptu-4m-1	speedup
ParallelDHL	1	252.13	1	135.79	1
	4	147.20	1.71	142.34	0.95
	8	102.83	2.45	145.04	0.93
	16	56.48	4.46	156.05	0.87
	24	48.19	5.23	172.81	0.78
RDFox	1	20.55	1	50.95	1
	4	56.48	0.36	72.42	0.7
	8	54.52	0.38	74.43	0.68
	16	60.98	0.34	74.23	0.69
	24	58.47	0.35	79.46	0.64

downtrend of reasoning times also exists when handling the datasets nptu-4m-i, $i \in \{1, 2, 4, 8, 16\}$ (see Figure 7(right)). Although the PTU and NPTU datasets are not real-world examples, these experiments indeed verify that MG-depths determines the reasoning time in parallel considering that the input ontology sizes are close.

We now discuss the issue of parallel tractability based on the experimental results of ptu-4m-1 and nptu-4m-1. For the dataset ptu-4m-1, the trends of reasoning times are different between ParallelDHL and RDFox. Table 6 shows that RDFox does not benefit from the allocation of several threads for ptu-4m-1; the speedups under different numbers of threads stay around 1. This means that reasoning times are not reduced with an increasing number of threads. ParallelDHL shows a better speedup effect. With only one allocated thread, ParallelDHL finishes the materialization of ptu-4m-1 in 135.79 seconds (Table 6). When four and more threads are allocated, the reasoning performance is improved significantly. The maximal speedup is 5.23 using 24 threads. Specially, with 16 or more threads, ParallelDHL requires less time to finish the materialization compared to RDFox. From Figure 8(left), there is an obvious downtrend of the reasoning times from one thread to 24 threads. The main reason for the difference between ParallelDHL and RDFox is that the PTU ontology is tractable in parallel and ParallelDHL is optimized based on SWD paths. The PTU ontology follows the simple-concept restriction. Thus, there is no twisting path in the materialization graph of ptu-4m-1 although all articles are involved in one reference chain. This allows for computing the relation S_{rch} only once in the second iteration of Step 2 of Algorithm A_{prc} when handling ptu-4m-1 (referring to the analysis for Example 7). It can also be

checked that all the axioms of the form $\texttt{CollegeArticle}(a_i)$ occur in an SWD path. Thus, the optimization based on SWD paths works when handling ptu-4m-1. For RDFox, since it is not optimized specially for this case, it performs the task of materialization over ptu-4m-1 similarly to that over nptu-4m-1; specifically, the axioms of the form $\texttt{CollegeArticle}(a_i)$ are derived sequentially. From Figure 7(left), we can see that the optimization used in ParallelDHL also leads to a better performance compared to RDFox when handling ptu-4m-1, ptu-4m-2 and ptu-4m-4 using 24 threads.

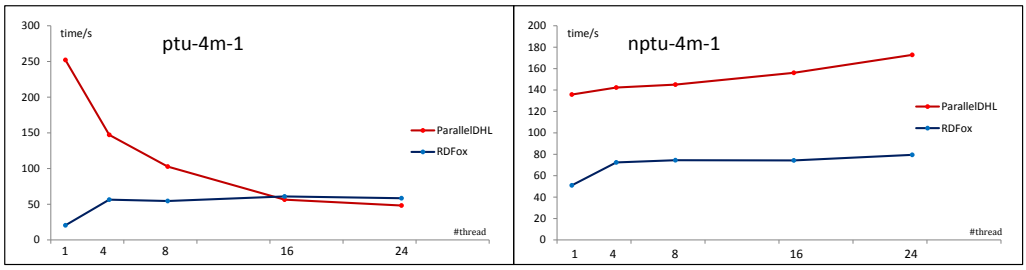

Figure 8: (left) the reasoning times over nptu-4m-1; (right) the reasoning times over ptu-4m-1.

From Figure 8(right), we can see that for both ParallelDHL and RDFox the reasoning efficiency is not improved when several threads are allocated. On the contrary, with less than 24 threads allocated, materialization takes less time. In detail, the reasoning time of RDFox using 24 threads is 79.46 seconds, while it is 50.95 seconds using only one thread. Further, the speedups of RDFox stay below 1 using more than 1 threads (see Table 6). This means that the reasoning times cannot be reduced with several threads being allocated. This maybe unexpected behavior is also shown by ParallelDHL and is caused by the issue of path twisting. The NPTU ontology does not follow the simple-concept restriction since the concepts `CollegeConference` and `CollegeSession` are not simple. The materialization of nptu-4m-i ($i \in \{1, 2, 4, 8, 16\}$) suffers from the issue of path twisting which is similar to the case in Example 6. Specifically, the axioms of the form $\texttt{CollegeArticle}(a_i)$ are the joint nodes in the twisted path. According to the analysis in Example 6, parallel computation can hardly handle this case. Moreover, the dataset nptu-4m-1 has only one article reference chain. This means that all the axioms of the form $\texttt{CollegeArticle}(a_i)$ actually appear in one path of the target materialization graph. Thus, they depend on each other and cannot be derived in parallel. On the other hand, the working mechanism of ParallelDHL and RDFox is based on a thread pool, where several threads are maintained and scheduled. The maintenance and scheduling of the thread pool also produces overheads, in particular, when the parallel computation cannot improve the total efficiency. This is why the materialization of nptu-4m-1 using 1 thread has a better performance.

7. Related Work

The parallel reasoner RDFox [4] is also used in the evaluation of our implementation. RDFox is a highly optimized system for reasoning over datalog rewritable ontology languages. Algorithm A_{bsc} proposed in Section 3 is similar to the main algorithm of RDFox (see [4], Sections 3 and 4). The main difference is that a group of rule instantiations is handled by one processor (namely a thread) in RDFox, while in Algorithm A_{bsc}, each rule instantiation is assigned to a

unique processor. Analogous to the method used in RDFox, ParallelDHL is also implemented by assigning a group of rule instantiations to a processor. Specially, ParallelDHL is also optimized by using SWD paths for handling ontologies that belong to parallel tractability classes. The experimental results show that this optimization results in a better performance on ontologies that are tractable in parallel compared with RDFox.

There is work that studies parallel reasoning in RDFS and OWL. The current methods mainly focus on optimizing reasoning algorithms from different aspects. Goodman et al. [10] propose a new kind of encoding method for RDF triples to achieve a high performance. This method can significantly yield a throughput improvement and optimize parallel RDF reasoning and query answering. Peters et al. [5] use the RETE algorithm to accelerate rule matching for RDFS reasoning. Subercaze et al. [6] propose a more efficient storage technique and optimize the join operations in parallel reasoning. The issue of balance distribution of parallel tasks is also studied [30, 27]. Two approaches are explored, i.e., *rule partitioning* (allocating parallel tasks to different processors based on rules) and *data partitioning* (allocating parallel tasks based on data). The evaluation results indicate that the efficiency of balance distribution varies with respect to different datasets. On the other hand, parallel reasoning is also implemented for OWL fragments, e.g., OWL RL [14], OWL EL [31], OWL QL [32], and even highly expressive languages [7, 33, 34, 8]. In current work, several techniques are proposed to adapt parallel computation to OWL reasoning tasks. A kind of graph-based method is discussed by Lembo et al. [32] to enhance OWL QL classification. Several pruning techniques are proposed to optimize the Tableaux algorithm [33, 34, 8]. The *lock-free technique* is applied by Kazakov et al. [31] and Steigmiller et al. [7].

Another line of optimizing parallel reasoning is to utilize high-performance computing platforms. For in-memory platforms, different supercomputers such as Cray XMT or Yahoo S4 have early on been used in parallel RDF reasoning [9, 10]. Heino and Pan [12] report on their work on RDFS reasoning based on massively parallel GPU hardware. Distributed parallel platforms such as MapReduce and Peer-to-Peer networks are also used for RDFS reasoning. The representative systems are WebPIE [11], Marvin [35] and SAOR [36]. Different techniques are discussed in this work to tackle the special problems in distributed computing. However, to study the issue of parallel tractability on distributed platforms, more issues have to be discussed, e.g., *network structures* and *communications*. This is not the focus in this paper.

Different from the above work, the purpose of this paper is to study the issue of the parallel tractability of materialization from the perspective of data. The results given in this paper guarantee the parallel tractability theoretically, regardless of what optimization techniques and platforms as discussed above are used.

8. Conclusions

In this paper, we studied the problem of parallel tractability of materialization on datalog rewritable ontologies. To identify the parallel tractability classes, we proposed two $\mathtt{NC}$ algorithms, Algorithm $\mathsf{A}^{\psi}_{\text{bsc}}$ and Algorithm $\mathsf{A}^{\psi}_{\text{opt}}$, that perform materialization on datalog rewritable ontology languages. Based on these algorithms, we identified the corresponding parallel tractability classes such that materialization on the datalog programs in these classes is in $\mathtt{NC}$ complexity. We further studied two specific ontology languages, DL-Lite and DHL (including one of its extension). We showed that any ontology expressed in DL-Lite$_{core}$ or DL-Lite$_{\mathcal{R}}$ is tractable in parallel. For DHL and DHL($\circ$), we proposed two restrictions such that materialization is tractable in parallel.

We analyzed the benchmark LUBM and the real-world dataset YAGO and gave the cases where these ontologies belong to parallel tractability classes. On the other hand, we used an optimization strategy based on SWD paths to give a practical algorithm variant Algorithm A_{prc}, which can also be restricted to an NC version. We implemented a system based on Algorithm A_{prc} and compared it to the state-of-the-art reasoner RDFox. The experimental results showed that the optimizations proposed in this paper result in a better performance on ontologies that are tractable in parallel compared to RDFox.

In future work, we plan to extend our work in two lines. One line is a further study of parallel tractability of other expressive ontology languages, in addition to datalog rewritable ontology languages. Since different expressive OWL languages have different syntaxes and higher reasoning complexities than PTime-complete, we need to explore more restrictions that are practical and make materialization tractable in parallel. Another line of work is to study how to further apply the results in practice. One idea is to apply the technique of SWD paths to enhance other reasoning-based tasks such as ontology classification, ontology debugging and query answering.

Acknowledgments

Guilin Qi was partially supported by NSFC grant 61672153 and the 863 program under Grant 2015AA015406.

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider, The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, 2003.

[2] O. Lehmberg, D. Ritze, R. Meusel, C. Bizer, A Large Public Corpus of Web Tables containing Time and Context Metadata, in: Proc. of WWW, 75–76, 2016.

[3] R. Meusel, C. Bizer, H. Paulheim, A Web-scale Study of the Adoption and Evolution of the schema.org Vocabulary over Time, in: Proc. of WIMS, 15:1–15:11, 2015.

[4] B. Motik, Y. Nenov, R. Piro, I. Horrocks, D. Olteanu, Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems, in: Proc. of AAAI, 129–137, 2014.

[5] M. Peters, S. Sachweh, A. Zündorf, Large Scale Rule-Based Reasoning Using a Laptop, in: Proc. of ESWC, 104–118, 2015.

[6] J. Subercaze, C. Gravier, J. Chevalier, F. Laforest, Inferray: fast in-memory RDF inference, J. PVLDB 9 (6) (2016) 468–479.

[7] A. Steigmiller, T. Liebig, B. Glimm, Konclude: System description, J. Web Sem. 27 (2014) 78–85.

[8] K. Wu, V. Haarslev, A Parallel Reasoner for the Description Logic ALC, in: Proc. of DL, 675–690, 2012.

[9] J. Hoeksema, S. Kotoulas, High-performance Distributed Stream Reasoning using S4, in: Proc. of OOR, 2011.

[10] E. L. Goodman, E. Jimenez, D. Mizell, S. Al-Saffar, B. Adolf, D. J. Haglin, High-Performance Computing Applied to Semantic Databases, in: Proc. of ESWC, 31–45, 2011.

[11] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, H. E. Bal, WebPIE: A Web-scale Parallel Inference Engine using MapReduce, J. Web Sem. 10 (2012) 59–75.

[12] N. Heino, J. Z. Pan, RDFS Reasoning on Massively Parallel Hardware, in: Proc. Of ISWC, 133–148, 2012.

[13] R. Greenlaw, H. J. Hoover, W. L. Ruzzo, Limits to Parallel Computation: P-Completeness Theory, Oxford University Press, New York, 1995.

[14] V. Kolovski, Z. Wu, G. Eadon, Optimizing Enterprise-Scale OWL 2 RL Reasoning in a Relational Database System, in: Proc. of ISWC, 436–452, 2010.

[15] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family, J. Autom. Reasoning 39 (3) (2007) 385–429.

[16] B. N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: combining logic programs with description logic, in: Proc. of WWW, 48–57, 2003.

[17] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[18] M. Krötzsch, S. Rudolph, P. Hitzler, Complexity Boundaries for Horn Description Logics, in: Proc. of AAAI, 452–457, 2007.

[19] Y. Kazakov, Consequence-Driven Reasoning for Horn SHIQ Ontologies, in: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, 2040–2045, 2009.

[20] H. J. ter Horst, Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary, J. Web Sem. 3 (2-3) (2005) 79–115.
[21] I. Horrocks, U. Sattler, Decidability of SHIQ with complex role inclusion axioms, J. Artif. Intell. 160 (1-2) (2004) 79–104.
[22] L. G. Valiant, A Bridging Model for Parallel Computation, Commun. ACM (1990) 103–111.
[23] H. J. Karloff, S. Suri, S. Vassilvitskii, A Model of Computation for MapReduce, in: Proc. of SODA, 938–948, 2010.
[24] E. Allender, Reachability Problems: An Update, in: Proc. of CiE, 25–27, 2007.
[25] P. Hayes, RDF Semantics, W3C Recommendation 10 February 2004, https://www.w3.org/TR/2004/REC-rdf-mt-20040210/ .
[26] B. Cuenca Grau, I. Horrocks, M. Krötzsch, C. Kupke, D. Magka, B. Motik, Z. Wang, Acyclicity Notions for Existential Rules and Their Application to Query Answering in Ontologies, J. Artif. Intell. 47 (2013) 741–808.
[27] J. Weaver, J. A. Hendler, Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples, in: Proc. of ISWC, 682–697, 2009.
[28] F. Mahdisoltani, J. Biega, F. M. Suchanek, YAGO3: A Knowledge Base from Multilingual Wikipedias, in: Proc. of CIDR, 2015.
[29] F. M. Suchanek, G. Kasneci, G. Weikum, YAGO: A Large Ontology from Wikipedia and WordNet, J. Web Sem. 6 (3) (2008) 203–217.
[30] R. Soma, V. K. Prasanna, A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases, in: Proc. of ISCA, 19–25, 2008.
[31] Y. Kazakov, M. Krötzsch, F. Simancik, The Incredible ELK - From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ Ontologies, J. Autom. Reasoning (2014) 1–61.
[32] D. Lembo, V. Santarelli, D. F. Savo, A Graph-Based Approach for Classifying OWL 2 QL Ontologies, in: Proc. of DL, 747–759, 2013.
[33] T. Liebig, F. Müller, Parallelizing Tableaux-Based Description Logic Reasoning, in: Proc. of OTM Workshops, 1135–1144, 2007.
[34] A. Schlicht, H. Stuckenschmidt, Distributed Resolution for ALC, in: Proc. of DL, 326–341, 2008.
[35] E. Oren, K. Spyros, A. George, S. Ronny, ten Teije Annette, van Harmelen Frank, Marvin: Distributed reasoning over large-scale Semantic Web data, J. Web Sem. (2009) 305–316.
[36] A. Hogan, A. Harth, A. Polleres, Scalable Authoritative OWL Reasoning for the Web, Int. J. Semantic Web Inf. Syst. 5 (2) (2009) 49–90.

Appendix

Appendix A. Proof of Lemma 1

Lemma 1 *Given a datalog program $P = \langle R, \boldsymbol{I} \rangle$, we have (1) A_{bsc} halts and returns a materialization graph $\mathcal{G}$ of P; (2) $\mathcal{G}$ has the the minimum depth among all the materialization graphs of P.*

Proof: (1) First, whenever a processor p adds a new node v to $\mathcal{G}$, it has to first check whether v has already been in $\mathcal{G}$ and does nothing if v is in $\mathcal{G}$. Thus $\mathcal{G}$ turns out to be an acyclic graph. Second, to show $\mathcal{G}$ is a complete materialization graph, we perform an induction on $T_R^{\omega}(\mathbf{I})$. Specifically, all the facts in $T_R^0(\mathbf{I})$ have to be in $\mathcal{G}$ by Step 1 of A_{bsc}. For $i > 0$, suppose that the ground atoms in $T_R^i(\mathbf{I})$ are in $\mathcal{G}$. It can be checked that the atoms in $T_R^{i+1}(\mathbf{I})$ have to be added to $\mathcal{G}$, since whenever a new node is derived from $T_R^i(\mathbf{I})$, there has to be a processor that would add it to $\mathcal{G}$.

(2) The *stage* (see the related contents in Section 2.3) of P is the lower bound of the depth of all materialization graphs. One can further check that, for the materialization graph $\mathcal{G}$ constructed by A_{bsc}, its depth is equal to the stage based on the induction above. Thus, $\mathcal{G}$ has the minimal depth among all the materialization graphs of P. □

Appendix B. Proof of Theorem 1

Theorem 1 *For any datalog program $P = \langle R, \boldsymbol{I} \rangle$, $P \in \mathcal{D}_{\mathsf{A}_{bsc}^{\psi}}$ iff P has a materialization graph whose depth is upper-bounded by $\psi(|\boldsymbol{I}|)$.*

Proof: We first prove that the number of iterations of Step 2 is actually the depth of the constructed materialization graph. We define the depths of nodes iteratively as follows: for each of the original node v, $\texttt{depth}(v)$=0; for each implicit node v' whose parents are $v_1, \ldots, v_i$, $\texttt{depth}(v')$=max{ $\texttt{depth}(v_1)$, …, $\texttt{depth}(v_i)$ } + 1. By performing an induction on the number of iterations of Step 2, one can check that an implicit node v' has to be added to $\mathcal{G}$ in the n^{th} iteration where n=$\texttt{depth}(v')$. Further, $\texttt{depth}(\mathcal{G})$=max{$\texttt{depth}(v_i)|v_i$ is in $\mathcal{G}$}. We then have that the number of iterations of Step 2 is $\texttt{depth}(\mathcal{G})$.

($\Rightarrow$) $P \in \mathcal{D}_{\mathsf{A}_{\text{bsc}}^{\psi}}$ means that $\mathsf{A}_{\text{bsc}}^{\psi}$ can return a materialization graph $\mathcal{G}$ of P. Recall that the number of iterations of Step 2 is bounded by $\psi(|\mathbf{I}|)$. Thus $\texttt{depth}(\mathcal{G})$ is also bounded by $\psi(|\mathbf{I}|)$.

($\Leftarrow$) Suppose P has a materialization graph $\mathcal{G}$ whose depth is upper-bounded by $\psi(|\mathbf{I}|)$. If $\mathcal{G}$ has the minimal depth among other materialization graphs of P, the number of iterations of Step 2 is also $\texttt{depth}(\mathcal{G})$ (Lemma 1) and, thus, upper-bounded by $\psi(|\mathbf{I}|)$. If A_{bsc} dose not return $\mathcal{G}$, then the returned graph $\mathcal{G}'$ should have a smaller depth compared with $\mathcal{G}$. In this case, this conclusion still holds. □

Appendix C. Proof of Lemma 2

Lemma 2 *Given a datalog program $P = \langle R, \boldsymbol{I} \rangle$, A_{opt} halts and outputs a materialization graph $\mathcal{G}$ of P.*

Proof: We prove the lemma in two stages: (1) the graph $\mathcal{G}$ returned by A_{opt} is a materialization graph; (2) $\mathcal{G}$ is a complete materialization graph. We first show that (1) holds by an induction on the iterations of Step 2 of A_{opt}.

Base case. Initially, $\mathcal{G}$ only contains all the original nodes. In this case, $\mathcal{G}$ is obviously a materialization graph.

Inductive case. According to the induction hypothesis, the partial graph constructed after the i^{th} ($i > 1$) iteration is a materialization graph, denoted by $\mathcal{G}^i$. We have to show that the partial graph constructed after the $(i+1)^{th}$ iteration is also a materialization graph, denoted by $\mathcal{G}^{i+1}$. The partial graph $\mathcal{G}^{i+1}$ is updated in A_{opt} by performing Step (**iii**) of Algorithm OPT. Suppose that $\texttt{rch}(B_k, H) \in S_{rch}$ is checked. It corresponds to the rule instantiation $B_1, .., B_k, .., B_n \rightarrow H$. Algorithm OPT next checks that there exists a node B' such that $\texttt{rch}(B', B_k) \in S^*_{rch}$ and B' is in $\mathcal{G}$. This means that B_k and H are derivable. The algorithm then adds new nodes B_k and H to $\mathcal{G}^i$ according to three cases. (1) If H is not in $\mathcal{G}^i$ while B_k is in $\mathcal{G}^i$, then H is added to $\mathcal{G}^{i+1}$, and the edges $e(B_1, H), \ldots, e(B_n, H)$ are created. (2) If neither of H and B_k is in $\mathcal{G}^i$, then H and B_k are added to $\mathcal{G}^{i+1}$, and the edges $e(B_1, H), \ldots, e(B_n, H)$ are also created. In the above two cases, H is a new node. Thus, $\mathcal{G}^{i+1}$ is acyclic. (3) If H is in $\mathcal{G}^i$ and H has no parent, B_k is added to $\mathcal{G}^{i+1}$, and the edges $e(B_1, H), \ldots, e(B_n, H)$ are also created. For the case that H is in $\mathcal{G}^i$ and H has parents, the algorithm does nothing. Thus, $\mathcal{G}^{i+1}$ is acyclic and satisfies the definition of materialization graph.

To show that (2) holds, we use the same method as in the proof of Lemma 1. We want to show that all the ground atoms in $T_R^{i+1}(\mathbf{I})$ have to be added to $\mathcal{G}$, with the induction hypothesis that the ground atoms in $T_R^i(\mathbf{I})$ are in $\mathcal{G}$. Suppose the ground atoms in $T_R^i(\mathbf{I})$ have been added to $\mathcal{G}$ by performing A_{opt}. For each atom α in $T_R^{i+1}(\mathbf{I})$, α actually has a special SWD path of length 1. This is because all parents of α have been in $\mathcal{G}$. According to the optimization strategy, α has to be added to $\mathcal{G}$ by applying Step 2 of A_{opt}. □

Appendix D. Proof of Lemma 3

Lemma 3 *For any DL-Lite$_{core}$ or DL-Lite$_{\mathcal{R}}$ ontology $\mathcal{O}$, there exists a materialization graph $\mathcal{G}$ such that each atom of the form $A(x)$ ($A \neq \bot$) or $R(x, y)$ in $\mathcal{G}$ has an SWD path.*

Proof: We first prove the lemma by an induction on applications of the datalog rules corresponding to DL-Lite$_{\mathcal{R}}$ axioms.

Base case. For each of the original node v of the form $A(x)$ ($A \neq \bot$) or $R(x, y)$, v has a special SWD path with v as the unique node. This conclusion holds for any materialization graph.

Inductive cases. For each datalog rule of the form '$B_1, \ldots, B_n \rightarrow H$' that is rewritten from some DL-Lite$_{\mathcal{R}}$ axiom, we have the induction hypothesis that $B_1, \ldots, B_n$ have SWD paths. We are left to prove that H has an SWD path as well in all materialization graphs of $\mathcal{O}$.

If H is in the form of $A(x)$ ($A \neq \bot$), it may be derived by applying (T1) and (T3). Thus, we conduct the induction by distinguishing these two cases as follows:

Case 1.1 $B \sqsubseteq A$. According to the induction hypothesis, node $B(x)$ has an SWD path, denoted by $(v_1, \ldots, v_n, B(x))$. We have that, in some materialization graph, node $A(x)$ has an SWD path of the form $(v_1, \ldots, v_n, B(x), A(x))$.

Case 1.2 $\exists R \sqsubseteq A$. Node $R(x, y)$ has an SWD path, denoted by $(v_1, \ldots, v_n, R(x, y))$, according to the induction hypothesis. It is obvious that node $A(x)$ has an SWD path of the form $(v_1, \ldots, v_n, B(x), R(x, y))$ in some materialization graph.

Since node $A(x)$ can only be derived in either Case 1.1 or Case 1.2, we have that node $A(x)$ has an SWD path in all materialization graphs of $\mathcal{O}$.

It is similar to prove the case where H is in the form of $R(x, y)$. Since node $R(x, y)$ may be derived by applying (R1) and (R2), we discuss these two cases.

Case 2.1 $S \sqsubseteq R$. According to the induction hypothesis, node $S(x, y)$ has an SWD path, denoted by $(v_1, \ldots, v_n, S(x, y))$. Obviously, node $R(x, y)$ has an SWD path of the form $(v_1, \ldots, v_n, S(x, y), R(x, y))$ in some materialization graph.

Case 2.2 $S \sqsubseteq R^-$. This case is similar to the above case.

For both Case 2.1 and Case 2.2, node $R(x, y)$ has an SWD path. Thus, node $R(x, y)$ has an SWD path in all materialization graphs of $\mathcal{O}$. We finish the proof for DL-Lite$_\mathcal{R}$. Since DL-Lite$_{\text{core}}$ is a subset of DL-Lite$_\mathcal{R}$ in terms of the expressivity, the above conclusion also applies to DL-Lite$_{\text{core}}$. □

Appendix E. Proof of Theorem 2

Theorem 2 *There exists a poly-logarithmically bounded function ψ s.t. $\mathcal{D}_{dl\text{-}lite} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

Proof: This theorem can be easily proved based on Lemma 3. Specifically, for any DL-Lite$_{\text{core}}$ or DL-Lite$_\mathcal{R}$ ontology $\mathcal{O}$, there always exists a poly-logarithmically bounded function ψ such that $\mathsf{A}^{\psi}_{\text{opt}}$ can handle the materialization of $\mathcal{O}$, since in any materialization graph, each node has an SWD path. More precisely, we can see that $\psi = 2$. □

Appendix F. Proof of Theorem 3

Theorem 3 *There exists a poly-logarithmically bounded function ψ s.t. $\mathcal{D}_{dhl} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

Proof: Observe that, for a datalog program that is transformed from a DHL ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, a rule with a binary atom as its head can only be of the form either (R1), (R2) or (R3). This also indicates that the materialization of a DHL ontology can be separated into two stages: in the first stage (Stage 1), all the rules of the forms (R1-R3) are exhaustively applied; the consequences of the first stage also serve as the facts in the second stage (Stage 2). The rules of the forms (T1-T3) are then applied in Stage 2. It is obvious that, if both of Stage 1 and Stage 2 can be handled by performing $\mathsf{A}^{\psi}_{\text{opt}}$ for some poly-logarithmical function ψ, then the whole materialization can be handled by $\mathsf{A}^{\psi}_{\text{opt}}$. In what follows, we investigate the above two stages, respectively, and show that, for any datalog program in $\mathcal{D}_{dhl}$, such a poly-logarithmical function ψ exists.

In Stage 1, the rules of the forms (R1-R3) are applied to add new nodes to the constructed materialization graph. We can observe that rule (R3) is used for computing the transitive closure over roles. As mentioned before, there exists an NC algorithm for computing the transitive closure. Inspired by this, we can prove that A^{ψ}_{opt} *handles Stage 1 for some poly-logarithmical function ψ*. The proof of this results can be shown by distinguishing simple roles and non-simple roles. For each role $R \in \mathbf{R}$, we further define a set δ_R, where $\mathbf{R}$ is the set of roles (role names and their inverses) in the considered ontology, as follows:

Definition 6. *Let $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ be a DHL ontology with $\boldsymbol{R}$ the set of roles. For each $R \in \boldsymbol{R}$, we define δ_R as the following set of assertions:*

$$\begin{aligned}\delta_R = &\{R(a,b) \mid R(a,b) \in \mathcal{A}\} \cup \\ &\{R(a,b) \mid R'(a,b) \in \mathcal{A} \text{ and } R' \sqsubseteq_* R\} \cup \\ &\{R(a,b) \mid R'(b,a) \in \mathcal{A} \text{ and } R' \sqsubseteq_* R^-\}.\end{aligned}$$

Let δ_R^* be the transitive closure of δ_R where R is a transitive role.[17] We then have the following lemma.

Lemma 4. *Let $P = \langle R, \boldsymbol{I} \rangle$ be the datalog program for an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$. We then have that $P \models R(a,b)$ implies: (1) if R is a simple role, $R(a,b) \in \delta_R$; (2) if R is a transitive role, $R(a,b) \in \delta_R^*$; (3) if R is a non-simple role but not a transitive role, then $R(a,b) \in \delta_R$, or there exists a transitive role R' such that $R' \sqsubseteq_* R$ and $R'(a,b) \in \delta_{R'}^*$.*

Note that, for all roles R, δ_R can be computed by only applying (R1) and (R2). In this sense, Lemma 4 also indicates that: (1) for each implicit node $R(a,b)$ where R is a simple role, it can be added to a materialization graph by only applying (R1) and (R2); (2) for each transitive role R, all implicit nodes of the form $R(a,b)$ are in the transitive closure δ_R^*, which can be computed by an NC algorithm on δ_R; (3) for each role R that is a non-simple role but not a transitive role, one can further perform (R1) and (R2) iteratively based on all transitive closures $\delta_{R'}^*$ where R' is a transitive role. Since all nodes generated by only applying (R1) and (R2) have SWD paths, the computations of (1) and (3) can be handled by $\mathsf{A}_{\text{opt}}^{\psi}$. Further, the transitive computation in part (2) can also be handled by $\mathsf{A}_{\text{opt}}^{\psi}$. Thus, there exists a poly-logarithmical function ψ such that $\mathsf{A}_{\text{opt}}^{\psi}$ handles Stage 1.

The results of Stage 1 serve as the facts of Stage 2. In other words, all binary atoms are the original nodes in Stage 2. This also means that the rules of form (T1) and (T3) generate nodes with SWD paths. Due to the simple-concept restriction, for each rule instantiation $A_1(a), A_2(a) \rightarrow B(a)$, either $A_1(a)$ or $A_2(a)$ always has an SWD path. Thus, Stage 2 can be handled by A_{opt} in at most two iterations (see Theorem 3). Based on the above analysis, this theorem holds.

We are now left to prove Lemma 4. We conduct the proof by an induction on the derivation of $P = \langle R, \mathbf{I} \rangle$. We distinguish inductive cases by different rules (R1-R3) that are possibly applied to derive $R(a,b)$.

Base case. If $R(a,b) \in \mathbf{I}$, $R(a,b) \in \delta_R$ holds. In this case, regardless of whether R is a non-simple or a simple role, all of (1), (2) and (3) in Lemma 4 hold.

Inductive case. We first analyze the case where R is a simple role.

Case 1.1 $R(a,b)$ is derived by applying the rule (R1), w.l.o.g., $R'(a,b) \rightarrow R(a,b)$, which also means that $R' \sqsubseteq R \in \mathcal{R}$. Since R is a simple role, R' has to be a simple role according to Definition 7. According to the induction hypothesis, $R'(a,b) \in \delta_{R'}$ (Case 1.1.1). If $R'(a,b) \in \mathcal{A}$, $R(a,b) \in \delta_R$ holds according to Definition 7 (Case 1.1.2). If there exists some role R'' such that $R''(a,b) \in \mathcal{A}$ and $R'' \sqsubseteq R'$, we also have $R(a,b) \in \delta_R$ since $R'' \sqsubseteq_* R$ holds (Case 1.1.3). If there exists some role R'' such that $R''(b,a) \in \mathcal{A}$ and $R'' \sqsubseteq R'^-$, the case is similar to Case 1.1.2.

[17] Note that, in DHL, a transitive role is a non-simple role, while a non-simple role may not be a transitive role.

Case 1.2 $R(a,b)$ is derived by applying the rule (R2), w.l.o.g., $R'(a,b) \rightarrow R(b,a)$, which also means that $R' \sqsubseteq R^- \in \mathcal{R}$. This case is similar to Case 1.1.

Case 1.3 $R(a,b)$ is derived by applying the rule (R3). This case is impossible, since R is a simple role.

We next analyze the case where R is a transitive role.

Case 2.1 $R(a,b)$ is derived by applying the rule (R1), w.l.o.g., $R'(a,b) \rightarrow R(a,b)$, which also means that $R' \sqsubseteq R \in \mathcal{R}$.

Case 2.1.1 R' is a simple role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$ holds. Similarly to Case 1.1.1, $R(a,b) \in \delta_R$ holds. Obviously, $R(a,b) \in \delta_R^*$ also holds.

Case 2.1.2 R' is a non-simple role. By the induction hypothesis, if $R'(a,b) \in \delta_{R'}$ holds, then $R(a,b) \in \delta_R$ also holds. On the other hand, if there exists a transitive role R'' (if R' is a transitive role, then $R'' \equiv R'$) such that $R'' \sqsubseteq_* R'$ and $R''(a,b) \in \delta_{R''}^*$. Since $\delta_{R''}^*$ is the transitive closure of $\delta_{R''}$, we then have that there must exist such atoms $(R''(a,c_1), R''(c_1,c_2), \ldots, R''(c_n,b))$ in $\delta_{R''}$. Further, due to $R'' \sqsubseteq_* R$ and Definition 7, we have that $R(a,c_1)$, $R(c_1,c_2), \ldots, R(c_n,b) \in \delta_R$. Thus, $R(a,b) \in \delta_R^*$ also holds.

Case 2.2 $R(a,b)$ is derived by applying the rule (R2), w.l.o.g., $R'(b,a) \rightarrow R(a,b)$, which also means $R' \sqsubseteq R^- \in \mathcal{R}$.

Case 2.2.1 R' is a simple role. By induction hypothesis, $R'(b,a) \in \delta_{R'}$. Similarly to Case 1.2, $R(a,b) \in \delta_R$ holds. Obviously, $R(a,b) \in \delta_R^*$ also holds.

Case 2.2.2 R' is a non-simple role. By the induction hypothesis, if $R'(b,a) \in \delta_{R'}$, then $R(a,b) \in \delta_R$ also holds. On the other hand, if there exists a transitive role R'' (if R' is a transitive role, then $R'' \equiv R'$) such that $R'' \sqsubseteq_* R'$ and $R''(b,a) \in \delta_{R''}^*$. Since $\delta_{R''}^*$ is the transitive closure of $\delta_{R''}$, we then have that there must exist such atoms $(R''(b,c_1), R''(c_1,c_2), \ldots, R''(c_n,a))$ in $\delta_{R''}$. Further, due to $R'' \sqsubseteq_* R^-$ and Definition 7, we have that $R(a,c_n)$, $R(c_n,c_{n-1}), \ldots, R(c_1,b) \in \delta_R$. Thus, $R(a,b) \in \delta_R^*$ also holds.

Case 2.3 $R(a,b)$ is derived by applying the rule (R3), w.l.o.g., $R(a,c), R(c,b) \rightarrow R(a,b)$, which also means that $R \circ R \sqsubseteq R \in \mathcal{R}$. By the induction hypothesis, $R(a,c), R(c,b) \in \delta_R^*$ holds. Obviously, $R(a,b) \in \delta_R^*$ also holds.

We finally study the case where R is a non-simple but not a transitive role.

Case 3.1 $R(a,b)$ is derived by applying the rule (R1), w.l.o.g., $R'(a,b) \rightarrow R(a,b)$, which also means that $R' \sqsubseteq R \in \mathcal{R}$.

Case 3.1.1 R' is a simple role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$ holds. Similarly to Case 1.1.1, $R(a,b) \in \delta_R$ holds. Obviously, $R(a,b) \in \delta_R^*$ also holds.

Case 3.1.2 R' is a non-simple role. By the induction hypothesis, if $R'(a,b) \in \delta_{R'}$, then $R(a,b) \in \delta_R$ also holds. On the other hand, if there exists a transitive role R'' (if R' is a transitive role, then $R'' \equiv R'$) such that $R'' \sqsubseteq_* R'$ and $R''(a,b) \in \delta^*_{R''}$. Since $R'' \sqsubseteq_* R$ and R'' is the transitive role, third consequence in this lemma is satisfied.

Case 3.2 $R(a,b)$ is derived by applying the rule (R2), w.l.o.g., $R'(a,b) \rightarrow R(b,a)$, which also means that $R' \sqsubseteq R^- \in \mathcal{R}$. This case is similar to Case 3.1.

Case 3.3 $R(a,b)$ is derived by applying the rule (R3). This is impossible, since R is not a transitive role. □

Appendix G. Proof of Theorem 4

Theorem 4 *There exists a poly-logarithmically bounded function ψ s.t. $\mathcal{D}_{dhl(\circ)} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$.*

Proof: The proof idea of this theorem is similar to that of Theorem 3. That is, we separate the materialization of DHL(∘) ontologies into two stages: in Stage 1, all the rules of the forms (R1-R4) are exhaustively applied; in Stage 2, the rules of the forms (T1-T2) are then applied while the results of Stage 1 serve as facts. Stage 2 is as same as that of materializing DHL. Thus we only consider Stage 1 here.

Our target is to show that $\mathsf{A}^{\psi}_{\text{opt}}$ handles Stage 1. To this end, we also distinguish all roles by whether they are simple or not. For DHL(∘), we have to consider complex RIAs as well. Thus, the set δ_R for each role $R \in \mathbf{R}$ should be re-defined. We give the following definition of *role sequence sets* which is used to further define the set δ_R.

Definition 7. *Let $O = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ be a DHL(∘) ontology with $\mathbf{R}$ the set of roles. For each $R \in \mathbf{R}$, the* set of role sequences *for R, $\mathcal{L}(R)$, is defined as follows:*

$$
\begin{aligned}
\mathcal{L}(R) = &\{R' \mid R' \sqsubseteq_* R\} \cup \\
&\{R'^{-} \mid R' \sqsubseteq_* R^-\} \cup \\
&\{L_1 L_2 \mid R_1 \circ R_2 \sqsubseteq R' \in \mathcal{R} \textit{ is not of form (R3)}, L_1 \in \mathcal{L}(R_1), L_2 \in \mathcal{L}(R_2) \textit{ and } R' \in \mathcal{L}(R)\} \cup \\
&\{L_2^- L_1^- \mid R_1 \circ R_2 \sqsubseteq R' \in \mathcal{R} \textit{ is not of form (R3)}, L_1 \in \mathcal{L}(R_1), L_2 \in \mathcal{L}(R_2) \textit{ and } R'^{-} \in \mathcal{L}(R)\}.
\end{aligned}
$$

In the above definition, for a role sequence $L = R_1 R_2, \ldots, R_n$, let $L^- = R_n^-, \ldots, R_2^- R_1^-$. We then give the following definition for δ_R.

Definition 8. *Let $O = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ be a DHL(∘) ontology with $\mathbf{RN}$ the set of role names. For each $R \in \mathbf{NR}$, we define δ_R as the following set of assertions:*

$$
\begin{aligned}
\delta_R = &\{R(a,b) \mid R(a,b) \in \mathcal{A}\} \cup \\
&\{R(a,b) \mid R'(a,b) \in \mathcal{A} \textit{ and } R' \sqsubseteq_* R\} \cup \\
&\{R(a,b) \mid R'(b,a) \in \mathcal{A} \textit{ and } R' \sqsubseteq_* R^-\} \\
&\{R(a,b) \mid R_0 R_1, \ldots, R_n \in \mathcal{L}(R), R_i(x_i, x_{i+1}) \in \mathcal{A}, i \in \{0,1,\ldots,n\}, x_0 = a, x_{n+1} = b\}.
\end{aligned}
$$

Lemma 5. *Let $P = \langle R, \boldsymbol{I} \rangle$ be the datalog program for a DHL(∘) ontology $O = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$. We then have that $P \models R(a,b)$ implies: (1) if R is a simple role, $R(a,b) \in \delta_R$; (2) if R is a transitive*

role, then $R(a,b) \in \delta^*_R$*; (3) if* R *is a non-simple but not transitive role, then there exists a role sequence* $R_0R_1,\ldots,R_n \in \mathcal{L}(R)$ *such that* $R_i(x_i,x_{i+1}) \in \delta_{R_i}$*, or,* $R_i(x_i,x_{i+1}) \in \delta^*_{R_i}$ *if* R_i *is a transitive role, where* $i \in \{0,1,\ldots,n\}$ *and* $x_0 = a, x_{n+1} = b$*.*

The set δ^*_R also denotes the transitive closure of δ_R where R is a transitive role. Note that, for all roles R, δ_R can be computed by applying (R1), (R2) and (R4). The above lemma says: (1) for each implicit node $R(a,b)$ where R is a simple role, it can be added to a materialization graph by only applying (R1), (R2) and (R4); (2) for transitive roles R, all implicit nodes are in the transitive closure δ^*_R, which can be computed by an NC algorithm on δ_R; (3) for each role R that is a non-simple but not a transitive role, one can further perform (R1), (R2) and (R4) iteratively based on all transitive closures $\delta^*_{R'}$ where R' is a transitive role. All nodes generated by applying (R1) and (R2) have SWD paths in the first iteration; at least one role from the left-hand side of (R4) is restricted to be a simple role. Thus, the computations of (1) and (3) can be handled by $\mathsf{A}^{\psi}_{\text{opt}}$. Further, the transitive computation in part (2) can also be handled by $\mathsf{A}^{\psi}_{\text{opt}}$. In summary, there exists a poly-logarithmical function ψ such that $\mathsf{A}^{\psi}_{\text{opt}}$ handles Stage 1.

We now prove Lemma 5. We conduct the proof by an induction on the derivation of $P = \langle R, \mathbf{I}\rangle$. We distinguish inductive cases by different rules (R1-R4) that are possibly applied to derive $R(a,b)$.

Base case. If $R(a,b) \in \mathbf{I}$, $R(a,b) \in \delta_R$ holds. In this case, regardless of whether R is a non-simple or a simple role, all of the consequences (1), (2) and (3) in Lemma 5 hold.

Inductive case. We first study the case where R is a simple role.

Case 1.1 $R(a,b)$ is derived by applying the rule (R1), w.l.o.g., $R'(a,b) \rightarrow R(a,b)$, which also means that $R' \sqsubseteq R \in \mathcal{R}$. Since R is a simple role, R' has to be a simple role according to Definition 9. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$ holds (Case 1.1.1). If $R'(a,b) \in \mathcal{A}$, $R(a,b) \in \delta_R$ holds according to Definition 9 (Case 1.1.2). If there exists some role R'' such that $R''(a,b) \in \mathcal{A}$ and $R'' \sqsubseteq R'$, we also have that $R(a,b) \in \delta_R$ because $R'' \sqsubseteq_* R$ holds (Case 1.1.3). If there exists some role R'' such that $R''(b,a) \in \mathcal{A}$ and $R'' \sqsubseteq R'^-$, the case is similar to Case 1.1.2 (Case 1.1.4). There exists $R_0R_1,\ldots,R_n \in \mathcal{L}(R')$, where $R_i(x_i,x_{i+1}) \in \mathcal{A}$ and $x_0 = a, x_{n+1} = b$ for $i \in \{0,1,\ldots,n\}$. According to Definition 8, $R_0R_1,\ldots,R_n \in \mathcal{L}(R)$ holds since $R' \sqsubseteq R \in \mathcal{R}$ (Case 1.1.5).

Case 1.2 $R(a,b)$ is derived by applying the rule (R2), w.l.o.g., $R'(a,b) \rightarrow R(b,a)$, which also means that $R' \sqsubseteq R^- \in \mathcal{R}$. Since R is a simple role, R' has to be a simple role according to Definition 9. By the induction hypothesis, $R'(b,a) \in \delta_{R'}$ (Case 1.2.1). If $R'(b,a) \in \mathcal{A}$, $R(a,b) \in \delta_R$ holds according to Definition 9 (Case 1.2.2). If there exists some role R'' such that $R''(b,a) \in \mathcal{A}$ and $R'' \sqsubseteq R'$, we also have that $R(a,b) \in \delta_R$ because $R'' \sqsubseteq_* R^-$ holds (Case 1.2.3). If there exists some role R'' such that $R''(a,b) \in \mathcal{A}$ and $R'' \sqsubseteq R'^-$, this case is similar to Case 1.2.2 (Case 1.2.4). There exists $R_0R_1,\ldots,R_n \in \mathcal{L}(R')$, where $R_i(x_i,x_{i+1}) \in \mathcal{A}$ and $x_0 = b, x_{n+1} = a$ for $i \in \{0,1,\ldots,n\}$. According to Definition 8, $R_n^-,\ldots,R_0^- \in \mathcal{L}(R)$ holds since $R' \sqsubseteq R^- \in \mathcal{R}$ (Case 1.2.5).

Case 1.3 $R(a,b)$ is derived by applying the rule (R4), w.l.o.g., $R_1(a,c_2), R_2(c_2,b) \rightarrow R(a,b)$, which also means that $R_1 \circ R_2 \sqsubseteq R \in \mathcal{R}$. Since R is a simple role, both of R_1 and R_2 are simple roles. By the induction hypothesis, for each $R_i(c_i,c_{i+1}), i \in \{1,2\}$, where

$c_1 = a, c_3 = b, R_i(c_i, c_{i+1}) \in \delta_{R_i}$ hold. We have a role sequence $L_i \in \mathcal{L}(R_i)$ that may be constructed according to the following cases:

Case 1.3.1 If there exists a role R_i' such that $R_i'(c_i, c_{i+1}) \in \mathcal{A}$ and $R_i' \sqsubseteq_* R_i$, then we have that $L_i = R_i'$.

Case 1.3.2 If there exists a role R_i' such that $R_i'(c_{i+1}, c_i) \in \mathcal{A}$ and $R_i' \sqsubseteq_* R_i^-$, then we have that $L_i = R_i^{'-}$.

Case 1.3.3 If there exists a role sequence $R_0R_1, \ldots, R_m \in \mathcal{L}(R')$ s.t. $R_j(x_j, x_{j+1}) \in \mathcal{A}$ and $x_0 = c_i, x_{n+1} = c_{i+1}$ for $i \in \{0, 1, \ldots, m\}$ and $R' \sqsubseteq_* R_i$, then $L_i = R_0R_1, \ldots, R_m$ holds.

Based on $L_i, i \in \{1, 2\}$, we have that $L_1L_2 \in \mathcal{L}(R)$. Further $R(a, b) \in \delta_R$ holds.

Case 1.4 $R(a, b)$ is derived by applying the rule (R3). This case is impossible, since R is a simple but not a transitive role.

We next study the case where R is a transitive role.

Case 2.1 $R(a, b)$ is derived by applying the rule (R1), w.l.o.g., $R'(a, b) \rightarrow R(a, b)$, which also means that $R' \sqsubseteq R \in \mathcal{R}$.

Case 2.1.1 R' is a simple role. By the induction hypothesis, $R'(a, b) \in \delta_{R'}$. Similarly to Case 1.1, $R(a, b) \in \delta_R$ holds and, obviously, $R \in \mathcal{L}(R)$ holds.

Case 2.1.2 R' is a non-simple but not a transitive role. By the induction hypothesis, there exists $R_0R_1, \ldots, R_n \in \mathcal{L}(R')$ such that $R_i(x_i, x_{i+1}) \in \delta_{R_i}$ or $R_i(x_i, x_{i+1}) \in \delta^*_{R_i}$, where $i \in \{0, 1, \ldots, n\}$ and $x_0 = a, x_{n+1} = b$. Since $R' \sqsubseteq R$ holds, $R_0R_1, \ldots, R_n \in \mathcal{L}(R)$ also holds.

Case 2.1.3 R' is a transitive role. By the induction hypothesis, there exists $R'(a, b) \in \delta^*_{R'}$. Further, let $R'(a, c_1), R'(c_1, c_2), \ldots, R'(c_n, b) \in \delta_{R'}$. Since $R' \sqsubseteq R \in \mathcal{R}$ holds, we also have that $R(a, c_1), R(c_1, c_2), \ldots, R(c_n, b) \in \delta_R$. Obviously $R(a, b) \in \delta^*_R$ holds.

Case 2.2 $R(a, b)$ is derived by applying the rule (R2), w.l.o.g., $R'(b, a) \rightarrow R(a, b)$, which also means $R' \sqsubseteq R^- \in \mathcal{R}$.

Case 2.2.1 R' is a simple role. By the induction hypothesis, $R'(b, a) \in \delta_{R'}$. Similarly to Case 1.2, $R(a, b) \in \delta_R$ holds and obviously, $R \in \mathcal{L}(R)$ holds.

Case 2.2.2 R' is a non-simple but not transitive role. By the induction hypothesis, there exists $R_0R_1, \ldots, R_n \in \mathcal{L}(R')$ such that $R_i(x_i, x_{i+1}) \in \delta_{R_i}$ or $R_i(x_i, x_{i+1}) \in \delta^*_{R_i}$, where $i \in \{0, 1, \ldots, n\}$ and $x_0 = b, x_{n+1} = a$. Since $R' \sqsubseteq R^-$ holds, $R_n^-, \ldots, R_0^- \in \mathcal{L}(R)$ also holds.

Case 2.2.3 R' is a transitive role. By the induction hypothesis, there exists $R'(b, a) \in \delta^*_{R'}$. Further, let $R'(b, c_1), R'(c_1, c_2), \ldots, R'(c_n, a) \in \delta_{R'}$. Since $R' \sqsubseteq R^- \in \mathcal{R}$, we also have that $R(a, c_1), R(c_1, c_2), \ldots, R(c_n, b) \in \delta_R$. Obviously $R(a, b) \in \delta^*_R$ holds.

Case 2.3 $R(a,b)$ is derived by applying the rule (R4), w.l.o.g., $R_1(a,c_2), R_2(c_2,b) \rightarrow R(a,b)$, which also means that $R_1 \circ R_2 \sqsubseteq R \in \mathcal{R}$. Further, R is a transitive role. According to the simple-role restriction, both of R_1 and R_2 are simple roles. This also means that R_1, R_2 are simple roles. Further, $R_1(a,c_2) \in \delta_{R_1}$ and $R_2(c_2,b) \in \delta_{R_2}$. This case is similar to Case 1.3.

Case 2.4 $R(a,b)$ is derived by applying the rule (R3), w.l.o.g., $R(a,c), R(c,b) \rightarrow R(a,b)$, which also means that $R \circ R \sqsubseteq R \in \mathcal{R}$. Since R is a non-simple role, by the induction hypothesis, $R(a,c), R(c,b) \in \delta_R^*$ holds. Obviously, $R(a,b) \in \delta_R^*$ holds.

We finally study the case where R is a non-simple but not a transitive role.

Case 3.1 $R(a,b)$ is derived by applying the rule (R1), w.l.o.g., $R'(a,b) \rightarrow R(a,b)$, which also means that $R' \sqsubseteq R \in \mathcal{R}$.

- Case 3.1.1 R' is a simple role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$ holds. Similarly to Case 1.1, $R(a,b) \in \delta_R$ holds and obviously, $R \in \mathcal{L}(R)$ holds.
- Case 3.1.2 R' is a non-simple but not a transitive role. By the induction hypothesis, there exists $R_0R_1, \ldots, R_n \in \mathcal{L}(R')$ such that $R_i(x_i, x_{i+1}) \in \delta_{R_i}$ or $R_i(x_i, x_{i+1}) \in \delta_{R_i}^*$, where $i \in \{0, 1, \ldots, n\}$ and $x_0 = a, x_{n+1} = b$. Since $R' \sqsubseteq R$, $R_0R_1, \ldots, R_n \in \mathcal{L}(R)$ also holds.
- Case 3.1.3 R' is a transitive role. By the induction hypothesis, there exists $R'(a,b) \in \delta_{R'}^*$. According to Definition 8, $R' \in \mathcal{L}(R)$ holds. Then, R' is actually the role sequence that satisfies the condition in this lemma.

Case 3.2 $R(a,b)$ is derived by applying the rule (R2), w.l.o.g., $R'(b,a) \rightarrow R(a,b)$, which also means that $R' \sqsubseteq R^- \in \mathcal{R}$.

- Case 3.2.1 R' is a simple role. By the induction hypothesis, $R'(b,a) \in \delta_{R'}$. Similarly to Case 1.2, $R(a,b) \in \delta_R$ holds and obviously, $R \in \mathcal{L}(R)$ holds.
- Case 3.2.2 R' is a non-simple but not a transitive role. By the induction hypothesis, there exists $R_0R_1, \ldots, R_n \in \mathcal{L}(R')$ such that $R_i(x_i, x_{i+1}) \in \delta_{R_i}$ or $R_i(x_i, x_{i+1}) \in \delta_{R_i}^*$, where $i \in \{0, 1, \ldots, n\}$ and $x_0 = b, x_{n+1} = a$. Since $R' \sqsubseteq R^-$, $R_n^-, \ldots, R_0^- \in \mathcal{L}(R)$ also holds.
- Case 3.2.3 R' is a transitive role. By the induction hypothesis, there exists $R'(b,a) \in \delta_{R'}^*$. Similarly to Case 3.1 (Case 1.1.3), we have that R'^- is actually the role sequence that satisfies the condition in this lemma.

Case 3.3 $R(a,b)$ is derived by applying the rule (R4), w.l.o.g., $R_1(a,c_2), R_1(c_2,b) \rightarrow R(a,b)$, which also means that $R_1 \circ R_2 \sqsubseteq R \in \mathcal{R}$ holds. Further, R is a non-simple but not transitive role. According to the simple-role restriction, at least one of R_1 and R_2 is a simple role. W.l.o.g., let R_2 be a simple role, which also means that R_2 is a simple role. Further, $R_2(c_2,b) \in \delta_{R_2}$ holds. For R_1, by the induction hypothesis, $R_1(a,c_2) \in \delta_{R_1}^*$ holds. On the other hand, a role sequence L_{R_2} exists in $\mathcal{L}(R_2)$ such that c_2 and b are the starting and ending individual, respectively. Further, we have that $R_1L_{R_2} \in \mathcal{L}(R)$ holds. This satisfies the second condition in Lemma 5.

Case 3.4 $R(a,b)$ is derived by applying the rule (R3), w.l.o.g., $R(a,c), R(c,b) \rightarrow R(a,b)$, which also means that $R \circ R \sqsubseteq R \in \mathcal{R}$. Since R is not a transitive role, this case is impossible. □

Appendix H. Proof of Theorem 5

Theorem 5 *For any DHL(∘) ontology $\mathcal{O}$, Algorithm A_{prc} halts and outputs a materialization graph $\mathcal{O}$.*

Proof: Similarly to the proof of Lemma 2, this theorem is proved in two stages: (1) the graph $\mathcal{G}$ returned by A_{prc} is a materialization graph; (2) $\mathcal{G}$ is a complete materialization graph. One can show that (1) holds by performing the same induction on the iterations of Step 2 of A_{opt} (see the proof of Lemma 2). This is because that such an induction is performed regardless of how the relation S_{rch} is computed.

To prove that (2) holds, we conduct the induction used in the proof for Lemma 1. We aim to show that all the ground atoms in $T_R^{i+1}(\mathbf{I})$ have to be added to $\mathcal{G}$, with the induction hypothesis that the ground atoms in $T_R^i(\mathbf{I})$ are in $\mathcal{G}$. Suppose the ground atoms in $T_R^i(\mathbf{I})$ have been added to $\mathcal{G}$ by performing A_{prc}. For each atom α in $T_R^{i+1}(\mathbf{I})$, there has to be a relation (β, α) in S_{rch} obtained in some iteration; further β is already in $\mathcal{G}$ according to the inductive hypothesis. Thus, α has to be added to $\mathcal{G}$ by applying Step 2 of A_{prc}. □

Appendix I. Proof of Theorem 6

Theorem 6 *For any DHL(∘) ontology $\mathcal{O}$ that follows the simple-concept and the simple-role restrictions, there exists a poly-logarithmically bounded function ψ, such that A_{prc}^{ψ} outputs a materialization graph of $\mathcal{O}$.*

Proof: The proof is analogous to the proof of Theorem 3 and Theorem 4. We separate the materialization of a DHL(∘) ontology into two stages: role materialization (rules of the forms (R1-R4) are exhaustively applied) and concept materialization (rules of the forms (T1-T3) are exhaustively applied). The difference is that we further separate the role materialization into two successive stages: *the simple role materialization* (Stage SRM) and *the non-simple role materialization* (Stage NSRM). Similarly, the concept materialization is also separated into two successive stages: *the simple concept materialization* (Stage SCM) and *the non-simple concept materialization* (Stage NSCM). Specifically, in Stage SRM, all atoms of the form $R(a, b)$ are derived where R is a simple role, while all atoms of the form $R(a, b)$ for the non-simple role R are derived in Stage NSRM. Stage SCM and Stage NSCM can be explained similarly.

We aim to prove that there always exists a poly-logarithmically bounded function ψ such that A_{prc}^{ψ} can handle each of the four stages, Stage SRM, Stage NSRM, Stage SCM and Stage NSCM. Based on the previous result, we have that there exists a poly-logarithmically bounded function ψ' such that $\mathsf{A}_{prc}^{\psi'}$ handles the whole materialization.

We first consider Stage SCM. Stage SCM (the simple concept materialization) is conducted on axioms of two forms $A \sqsubseteq B$ and $\exists R.A \sqsubseteq B$, which correspond to datalog rules of the forms $A(x) \rightarrow B(x)$ and $R(x, y), A(y) \rightarrow B(x)$, respectively. Since role assertions are supposed to be fixed during Stage SCM, the role R in each rule of the form $R(x, y), A(y) \rightarrow B(x)$ can be viewed as an EDB predicate. This further means that, for each atom of the form $A(a)$ where A is a simple concept, $A(a)$ has an SWD path in the first iteration of A_{prc}. Recall Algorithm Prc. We use an induction to show that such an SWD path of $A(a)$ can be determined by S_{rch}. The atom $A(a)$ can be derived through either of the following rule instantiations:

Case 1 $B(a) \rightarrow A(a)$, where $\mathcal{O} \models B \sqsubseteq A$. We have that $\texttt{rch}(B(a), A(a))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of the SWD path for $B(a)$ can be determined. Thus, the existence of the SWD path of $A(a)$ can also be determined by S_{rch}.

Case 2 $R(a, b), B(b) \rightarrow A(a)$ where $\exists R.B \sqsubseteq A \in \mathcal{T}$. According to Algorithm Prc and the fact that $R(a, b)$ has to be derived, we have that $\texttt{rch}(B(b), A(a))$ is in S_{rch}. By the induction hypothesis, the existence of the SWD path for $B(b)$ can be determined. Thus, the existence of the SWD path of $A(a)$ can also be determined.

We next discuss the non-simple concept materialization (Stage NSCM), which is conducted on axioms of forms (T1-T3). According to the simple-concept restriction, for each axiom of the form $A_1 \sqcap A_2 \sqsubseteq B$, one of A_1 and A_2 should be a simple concept and can be viewed as an EDB predicate, since Stage SCM is completed. In Stage NSCM, for each atom of the form $A(a)$ where A is not a simple concept, $A(a)$ has an SWD path in the first iteration of A_{prc} due to the simple concept restriction. We also use an induction to show that such an SWD path of $A(a)$ can be determined by distinguishing different rule instantiations. One can refer to the cases of rule instantiations of the form $B(a) \rightarrow A(a)$ and $R(a, b), B(b) \rightarrow A(a)$ as Case 1 and Case 2, respectively. We only consider the case of the rule instantiation of the form $A_1(a), A_2(a) \rightarrow A(a)$ here.

Case 3 $A_1(a), A_2(a) \rightarrow A(a)$ where $A_1 \sqcap A_2 \sqsubseteq A \in \mathcal{T}$. If $A_1(a)$ is derived where A_1 is a simple concept, we have that $\texttt{rch}(A_2(a), A(a))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of an SWD path for $A_2(a)$ can be determined. Thus, the existence of the SWD path of $A(a)$ can also be determined. The case where $A_2(a)$ is derived and A_2 is a simple concept is similar.

The simple-role materialization (Stage SRM) can simply be conducted by checking the transitive closure of role inclusions, i.e., axioms of the forms (R1-R2). It is easy to check that the simple-role materialization is finished after the first iteration of A_{prc}.

We finally discuss Stage NSRM. For each atom of the form $R(a, b)$ where R is a non-simple role but not a transitive role, $R(a, b)$ has an SWD path in the first iteration of A_{prc} due to the simple role restriction. We use an induction to show that such an SWD path of $R(a, b)$ can be determined by distinguishing different rule instantiations as follows.

Case 4 $S(a, b) \rightarrow R(a, b)$ where $\mathcal{O} \models S \sqsubseteq R$. We have that $\texttt{rch}(S(a, b), R(a, b))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of an SWD path for $S(a, b)$ can be determined. Thus, the existence of an SWD path of $R(a, b)$ can also be determined.

Case 5 $S(b, a) \rightarrow R(a, b)$ where $\mathcal{O} \models S \sqsubseteq R^{-}$. We have that $\texttt{rch}(S(b, a), R(a, b))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of an SWD path for $S(b, a)$ can be determined. Thus, the existence of an SWD path of $R(a, b)$ can also be determined.

Case 6 $R_1(a, c), R_2(c, b) \rightarrow R(a, b)$ where $R_1 \circ R_2 \sqsubseteq R \in \mathcal{R}$. If $R_1(a, c)$ is derived where R_1 is a simple role, we have that $\texttt{rch}(R_2(c, b), R(a, b))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of an SWD path for $R_2(c, b)$

can be determined. Thus, the existence of an SWD path of $R(a, b)$ can also be determined. The case where $R_2(c, b)$ is derived and R_2 is a simple role can be similarly proved.

We finally consider the derivation of atoms of the form $R(a, b)$, where R is a transitive role. From Lemma 5 we have that, $R(a, b) \in \delta_R$ or $R(a, b) \in \delta_R^*$ holds. According to the simple role restriction, for each sub-role S of R, S is a simple role or there exist axioms of the form $S_1 \circ S_2 \sqsubseteq S$ where S_1 and S_2 are simple roles. One can check that the set δ_R can be computed by A_{prc} in at most two iterations (see Cases 4, 5 and 6). We have to prove that the transitive closure δ_R^* can also be computed by A_{prc}^{ψ} for some poly-logarithmically bounded function ψ.

Case 7 $R(a, c), R(c, b) \rightarrow R(a, b)$ where R is a transitive role. One can construct a binary tree t where the root node is $R(a, b)$, the leaves are in δ_R and the height of t is upper-bounded by $\log(|\delta_R|)$. It can be checked that, A_{prc} can generate all nodes in each level of t in one iteration from the bottom. Thus, $R(a, b)$ can be derived in at most $\log(|\delta_R|)$ iterations. □