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] *Institution of Artificial Intelligence, University of Ulm*

Abstract

Materialization is an important reasoning service for applications built on the ontology languages, such as OWL. With a rapid growth of semantic data, it is challenging to perform materialization on large-scale ontologies efficiently. Current work proposes parallel algorithms and employs computing platforms of high-performance to make materialization sufficiently efficient and scalable in practice. However, there do exist ontologies, for which materialization cannot always be improved by parallelism (we use experiments to show this in Section 6). On the other hand, some well-known large-scale ontologies, such as YAGO, have been shown to have good performance for parallel reasoning, but they are expressed in ontology languages that are not parallelly tractable in theory, i.e., the efficiency of reasoning may not be improved on a parallel implementation. This motivates us to study the problem of *parallel tractability* of ontology materialization from a theoretical perspective. That is, we aim to identify the ontologies for which materialization is parallelly tractable, i.e., in NC complexity. The results of the parallel tractability can be further used to optimize materialization algorithms and to guide ontology engineers in creating parallelly tractable ontologies.

In this work, we focus on datalog rewritable ontology languages. We propose some algorithms, called NC algorithms, to identify several classes of datalog rewritable ontologies (called *parallelly tractable classes*) such that materialization over them is parallelly tractable. We then investigate the parallel tractability of materialization over two datalog rewritable ontology languages DL-Lite and DHL (*Description Horn Logic*). We show that materialization over any ontology expressed in DL-Lite$_{core}$ or DL-Lite$_{\mathcal{R}}$ is parallelly tractable. For DHL, there exist ontologies where materialization can hardly be parallelized. We propose to restrict the usage of DHL such that the restricted ontologies are parallelly tractable. We further extend the results to an extension of DHL that also allows *complex role inclusion axioms*. To verify the practical usability of the above results, we analyze real-world datasets and show that many ontologies expressed in DHL or its extension belong to the parallelly tractable classes. Based on the results of parallel tractability, we further implement an optimized parallel algorithm and compare it to the state-of-the-art reasoner RDFox. The experimental results show that the optimization used in our implementation results in a better performance on parallelly tractable ontologies compared with RDFox.

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

1. Introduction

The Web Ontology Language (OWL)[1] is an important standard for ontology language in the Semantic Web and knowledge-based applications. In these applications, *materialization*, which is the reasoning task of computing all implicit facts that follow from a given ontology [1], plays an important role. By means of it, developers and users can optimize query answering, ontology diagnosis and debugging. Since huge 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.

There has been work proposing parallel reasoning algorithms and employing parallel computing platforms for the ontology language RDFS and its extended fragments [4, 5, 6]. Several optimization 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 of highly expressive ontology languages [7, 8]. On the other hand, several work utilizes different kinds of computing platforms to make reasoning tasks more efficient in parallel, such as supercomputers [9, 10], MapReduce [11] and GPU servers [12].

The above work verifies the effectiveness of parallel reasoning based on the evaluations on different test datasets. However, for most popular ontology languages, even RDFS and datalog rewritable ontology languages, which have `PTime`-complete or higher complexity of reasoning in the worst case, they are not parallelly tractable according to [13], i.e., the efficiency of reasoning may not be improved on a parallel implementation. A possible reason of the high performance of parallel reasoning is that the utilized test datasets do not fall in the worst cases in terms of computational complexity. It has also been shown that some well-known large-scale ontologies, such as YAGO, have good performance for parallel reasoning [14], but they are expressed in ontology languages that are not parallelly tractable in theory. On the other hand, there do exist ontologies for which materialization cannot always be improved by parallelism (we use experiments to show this in Section 6). The current 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 parallelly tractable. These properties can be further used to optimize parallel algorithms and to guide ontology engineers in creating ontologies for which parallel tractability can be guaranteed theoretically.

According to [4], many real large-scale ontologies are essentially expressed in the ontology languages that can be rewritten into datalog rules. Thus, we focus on such datalog rewritable ontology languages in this paper. The main target of this paper is to identify the classes of datalog rewritable ontologies such that materialization over these ontologies is parallelly tractable, i.e., in 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 the NC class, one can give an *NC algorithm* that handles this problem in 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 has `PTime`-complete complexity. Thus, we first give NC algorithms that perform materialization, and then identify the

*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/

corresponding classes of datalog rewritable ontologies (called *parallelly tractable 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 to clarify how to apply the above NC algorithms to study the parallel tractability of ontology materialization. We show that DL-Lite$_{core}$ and DL-Lite$_{\mathcal{R}}$ are parallelly tractable. We then study the ontology language Description Horn Logic (DHL) [15], 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 well-known benchmarks and real-world datasets and show that many real ontologies following the proposed restrictions belong to the parallelly tractable classes. We implement a system based on an optimized NC algorithm for DHL materialization. We compare our system to the state-of-the-art reasoner RDFox. The experimental results show that the optimization proposed in this paper results in a better performance on parallelly tractable ontologies compared with RDFox.

The rest 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 the parallelly tractable materialization of DL-Lite, DHL and an extension of DHL in Section 4 respectively. In Section 5, we discuss how to optimize and implement the given NC algorithms. We analyze real-world datasets and evaluate our implementation in Section 6. We then discuss related work in Section 7 and conclude in Section 8. The detailed proofs of the theorems and the 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 [16], a *term* is a variable or a constant. An *atom* A is defined by $A \equiv p(t_1, ..., t_n)$ where p is a *predicate* (or *relational*) name, $t_1, ..., t_n$ 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: '$B_1, ..., B_n \rightarrow H$',[2] where H is referred to as the *head atom* and $B_1, ..., B_n$ 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 consists of rules and facts. 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] 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)	$R(a, b)$	$R(a, b)$

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 *structure transformation* techniques [17, 18].

DHL (short for *description horn logic*) [15] 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 either concept names, *top concept* ($\top$) or *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 *nominal*, *number restriction* and right-hand *existential restriction* ($A \sqsubseteq \exists R.B$). Second, the expressivity of DHL covers that of RDFS to some extent [15]. Reasoning with RDFS ontologies is `NP`-complete [19] and, thus, is not parallelly tractable. However, by applying some simplifications and restrictions, RDFS ontologies can be expressed in DHL [15] that has `PTime`-complete complexity for materialization.

In the initial work of DHL [15], *complex role inclusion axioms* (complex RIAs) of the form $R_1 \circ ... \circ R_n \sqsubseteq R$ are not considered, although they can be naturally transformed to 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 the 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 through knowledge and underpins the OWL profile OWL QL [20]. DL-Lite$_{core}$ and DL-Lite$_{\mathcal{R}}$ are the two basic fragments in DL-Lite. DL-Lite$_{core}$ requires that an TBox 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 RBox is not involved. DL-Lite$_{\mathcal{R}}$ is an extension of DL-Lite$_{core}$, 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$_{core}$ or DL-Lite$_\mathcal{R}$ can be transformed to a datalog program (see the corresponding rules in the right column of Table 1). Note that, for an axiom of the form (T4) $A \sqsubseteq \exists R$, it should be transformed to a first-order logic rule '$A(x) \rightarrow \exists y(R(x, y))$' where y is called a *free variable* (it is not occurring in the head atom $A(x)$). In this work, such a free variable y is eliminated via Skolemisation into a new individual o_R^A that corresponds to concept A and role R. One can check that the elimination of free variables does not influence the results of materialization by according to [20].

In what follows, for an ontology $O = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, we also use $P = \langle R, \mathbf{I} \rangle$ to represent the corresponding datalog program where R is the set of rules transformed from the axioms in $\mathcal{T}$ and $\mathcal{R}$, $\mathbf{I}$ 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 $R_1^- \sqsubseteq_* R_2^-$. In this paper, we also use the notion of *simple role*, which is 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 $P = \langle R, \mathbf{I} \rangle$, let $T_R(\mathbf{I}) = \{H\theta | \forall B_1, ..., B_n \rightarrow H \in R, B_i\theta \in \mathbf{I}(1 \leq i \leq n)\}$, 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 NC Complexity

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 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, the NC problems are also highly parallel feasible for other parallel models like BSP [22] and MapReduce [23]. The NC complexity is 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 should be 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 parallelly tractable. Since we assume that for any class of datalog programs where the rule set of each datalog program is fixed, the materialization problem is thus in data complexity PTime-complete, which is considered to be inherently sequential in the worst case [13]. In other words, the materialization problem on general datalog programs cannot be solved in parallel poly-logarithmic time unless P=NC. Thus, we say that *materialization on a class of datalog programs is parallelly tractable if there exists an algorithm that handles this class of datalog programs and runs in parallel poly-logarithmic time (this 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 Tractable Classes

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

Definition 1. ***(Parallelly Tractable Class)*** *Given a class $\mathcal{D}$ of datalog programs where the same rule set is shared, we say that $\mathcal{D}$ is a parallelly tractable datalog program (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 parallelly tractable ontology (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 give our 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 graph*. It makes the analysis of the given algorithm convenient.

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

- *H has a parent node or a child node;*
- *H is the original fact of P if H has the in-degree of* 0*;*

- $B_1, ..., B_n \rightarrow H \in P^*$ *is satisfied if* $e(B_1, H), ..., e(B_n, H) \in E$ *and* $B_1, ..., B_n$ *are all the parents of* H.

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, the nodes whose in-degree is 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 $\texttt{depth}(\mathcal{G})$, 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* 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 *is 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 whose in-degree is 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*.*

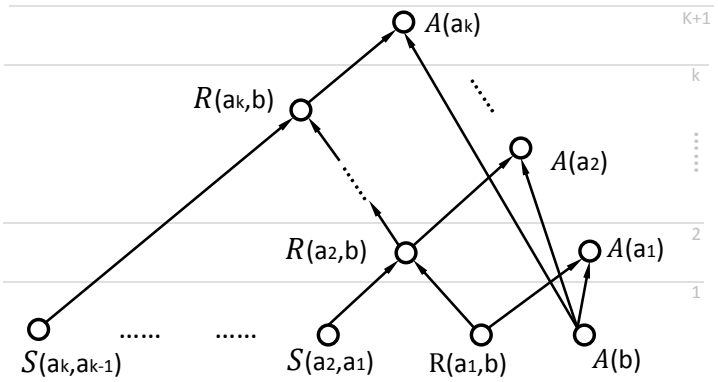

Figure 1: An example of a materialization graph.

We say that a materialization graph $\mathcal{G}$ is a *complete materialization graph* when $\mathcal{G}$ contains all ground atoms in $T_R^\omega(\mathbf{I})$. The set of nodes in a complete materialization graph is actually the result of materialization. Thus, the procedure of materialization can be transformed to the construction of a complete materialization graph. We pay our attention to complete materialization graphs and do not distinguish it to the notion 'materialization graph'. 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 (paral-

lel, random-access machine) that is mostly used to analyze the parallel complexity. This model allows us giving *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 access the state of an atom occurring in this rule instance in constant time. Algorithm A_{bsc} is then given as follows.

`Algorithm` A_{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 $B_1, ..., B_n \rightarrow H$, if the body atoms are all in $\mathcal{G}$ while H is not in $\mathcal{G}$,[3] the corresponding processor adds H to $\mathcal{G}$ and creates edges pointing from $B_1, ..., B_n$ to H.

(*Step 3*) If no processor can add more nodes and edges to $\mathcal{G}$, terminate, otherwise iterate 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 $(A(b), R(a_1, b), S(a_2, a_1), ..., S(a_k, a_{k-1}))$ 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 A_{bsc} and that, for any datalog program P, Algorithm A_{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 equals 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 EDB predicate, the maximum number of variables in any datalog rule,

[3]Suppose that each processor can use $O(1)$ time units to access the state of ground atoms, i.e., whether this ground atom has been added to the materialization graph. This can be implemented by maintaining an index of polynomial size.

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 occupies 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 $c_1 + l \cdot c_2$ where l denotes the number of iterations of Step 2.

When we say that an algorithm is an NC algorithm. It 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 use $\psi(|\mathbf{I}|)$ to bound the number of iterations of Step 2, then the computing time of Algorithm A_{bsc} is $c_1 + \psi(|\mathbf{I}|) \cdot c_2$. It is now an NC algorithm. We use A^{ψ}_{bsc} to denote the NC version of Algorithm A_{bsc}.

Based on A^{ψ}_{bsc}, we can identify a class of datalog programs $\mathcal{D}_{\mathsf{A}^{\psi}_{bsc}}$ such that all the datalog programs in it can be handled by A^{ψ}_{bsc}. It is obvious that $\mathcal{D}_{\mathsf{A}^{\psi}_{bsc}}$ 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 $\mathcal{D}_{\mathsf{A}^{\psi}_{bsc}}$.

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

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 $\mathbb{P}_{ex_1}$, 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 A^{ψ}_{bsc} is restricted in the sense that it cannot even work on the rather simple datalog program class $\mathbb{P}_{ex_1}$. It can be checked that, for some materialization graph $\mathcal{G}_{ex_1}$ corresponding to a datalog program in $\mathbb{P}_{ex_1}$, $\mathtt{depth}(\mathcal{G}_{ex_1}) = k$ 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 $\mathbb{P}_{ex_1}$ are polynomial of k. Thus, for any ψ that is poly-logarithmically bounded, we can always find a k large enough such that A^{ψ}_{bsc} terminates without constructing a materialization graph for each datalog program in $\mathbb{P}_{ex_1}$. However there indeed exists an NC algorithm that can handle $\mathbb{P}_{ex_1}$. We discuss this in the next part.

3.4. Optimizing Algorithm A_{bsc} via the 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.

Example 3. *Consider a 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*

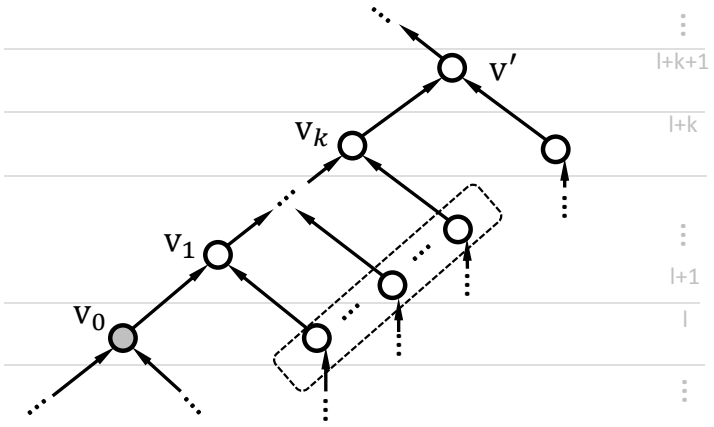

Figure 2: A partial materialization graph.

that have been added to $\mathcal{G}$. In this snapshot, v_0 has been newly added to $\mathcal{G}$ in the l^{th} ($l \geq 1$) iteration. Each of the nodes v_i ($1 \leq i \leq k$) has almost one parent node being not in $\mathcal{G}$, while v' has two parent nodes being 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 follows the condition: *if the parent node v_{i-1} is derivable, the child node v_i has to be derivable* (this is because that, for node v_i, 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 $\tau = (v_0, v_1, ..., v_k)$ where each edge (v_{i-1}, v_i) ($1 \leq i \leq k$) follows 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.

Backing to Example 3, since there exists an SWD path $(v_0, ..., v_i)$ for each node v_i ($1 \leq i \leq k$), 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 being 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 a $\texttt{rch}$ relation (denoted by S_{rch}) by performing the following process:

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

1. *if the body atoms $B_1, ..., B_n$ have all been added to $\mathcal{G}$, put $\texttt{rch}(B_1, H), ..., \texttt{rch}(B_n, H)$ 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 optimization that, for a node v, if there is a relation $\texttt{rch}(v', v) \in S^*_{rch}$ where v' has been added to $\mathcal{G}$, v has an SWD path and 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 a `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 $e(B_1, H), ..., e(B_n, H)$ 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. 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 $\{\texttt{rch}(R(a_{i-1}, b), R(a_i, b))|2 \leq i \leq k\} \cup \{\texttt{rch}(R(a_i, b), A(a_i))|1 \leq i \leq k\}$. In the transitive closure of S_{rch}, one can check that $\texttt{rch}(R(a_1, b), R(a_i, b))$, $\texttt{rch}(R(a_1, b), A(a_i)) \in S^*_{rch}(2 \leq i \leq k)$. 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 obtain an NC variant of A_{opt} analogously to the process for Algorithm A_{bsc}. It can be checked that an iteration of Step 2 in A_{opt} costs poly-logarithmic time, since the main part is computing S^*_{rch} by an NC algorithm. Thus, if the number of iterations of Step 2 is upper-bounded by a poly-logarithmical function, A_{opt} is an NC algorithm. Analogously to A^{ψ}_{bsc}, we use A^{ψ}_{opt} to

denote an NC variant. Specifically, for any datalog program P, the number of iterations of Step 2 in A_{opt} is bounded by $\psi(|P|)$, where ψ is a poly-logarithmically bounded function.

Based on A^{ψ}_{opt}, we can identify a PTD class $\mathcal{D}_{\mathsf{A}^{\psi}_{opt}}$. Further, we have the following corollary which implies that A^{ψ}_{opt} performs better than A^{ψ}_{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 $P \in \mathcal{D}_{\mathsf{A}^{\psi}_{bsc}}$. According to Theorem 1, the depth of the materialization graph $\mathcal{G}$ constructed by A^{ψ}_{bsc} is upper-bounded by $\psi(|P|)$. It is obvious that the number of nodes in each path of $\mathcal{G}$ is also upper-bounded by $\psi(|P|)$. 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 $\psi(|P|)$; 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 with $\mathcal{G}$. □

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 DL-Lite$_{core}$ or DL-Lite$_{\mathcal{R}}$ ontology $\mathcal{O}$, there exists a poly-logarithmically bounded function ψ such that Algorithm A^{ψ}_{opt} can handle materialization of $\mathcal{O}$. However, for DHL and DHL(∘), there exist ontology classes such that Algorithm A^{ψ}_{opt} does not work. We illustrate the reason why Algorithm A^{ψ}_{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$_{core}$ 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 $\mathcal{O}_{ex_5}$ where its 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 $\mathcal{O}_{ex_5}$ by $P_{ex_5} = \langle R, \boldsymbol{I}\rangle$, where 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).*

Consider performing Algorithm A_{opt} on the ontology $\mathcal{O}_{ex_5}$ in Example 5. (I) First, Algorithm A_{opt} adds all ABox assertions (only $Q(a, b)$ here) to $\mathcal{G}_{ex_5}$ that is initially an empty graph (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 $\mathcal{G}_{ex_5}$ 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 $\mathcal{G}_{ex_5}$ 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 $\mathcal{G}_{ex_5}$; if $A(a)$ is already in $\mathcal{G}_{ex_5}$, $B_1(a)$ is added to $\mathcal{G}_{ex_5}$ by creating an edge pointing from $A(a)$ to $B_1(a)$;

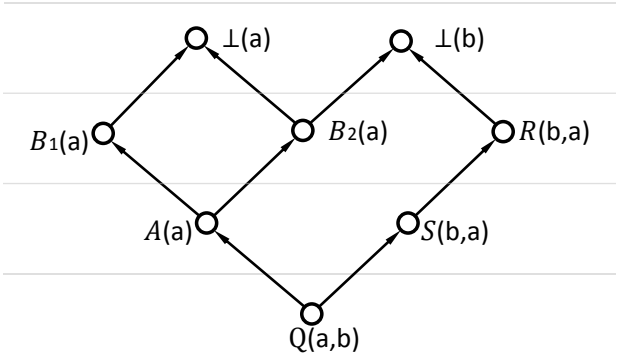

Figure 3: The materialization graph of $\mathcal{O}_{ex_5}$.

if $A(a)$ has not been added to $\mathcal{G}_{ex_5}$, Algorithm A_{opt} adds $A(a)$ (resp., $B_1(a)$) to $\mathcal{G}_{ex_5}$ and creates an edge pointing from $Q(a, b)$ to $A(a)$ (resp., an edge pointing from $A(a)$ to $B_1(a)$).[4] The 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 $\mathcal{G}_{ex_5}$ 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 satisfies any DL-Lite$_{core}$ 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 $\mathcal{G}_{ex_5}$ of Example 5). On the other hand, according to the syntaxes of DL-Lite, $\bot$ dose not occur on the left hand of any axiom. In other words, an atom of the form $\bot(x)$ cannot be the parent of any other node. This allows all atoms of the form $\bot(x)$ being added 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 A^{ψ}_{opt} can handle the materialization of $\mathcal{O}$; more precisely, we can set that $\psi = 2$. This result is consistent with that in [20]. Formally, we use $\mathcal{D}_{dl\_lite}$ to denote the set of all DL-Lite$_{core}$ and DL-Lite$_{\mathcal{R}}$ ontologies, and give the following theorem.

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

4.2. Parallelly Tractable Materialization of DHL

In this part, we study whether Algorithm A^{ψ}_{opt} can handle DHL ontologies. Unfortunately there exist DHL ontology classes such that Algorithm A^{ψ}_{opt} does not work for any poly-logarithmically bounded function ψ. In the following, we first give such a case to illustrate

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

the reason why Algorithm A^{ψ}_{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-logarithmical number of iterations of Step 2. We use the following example to illustrate it.

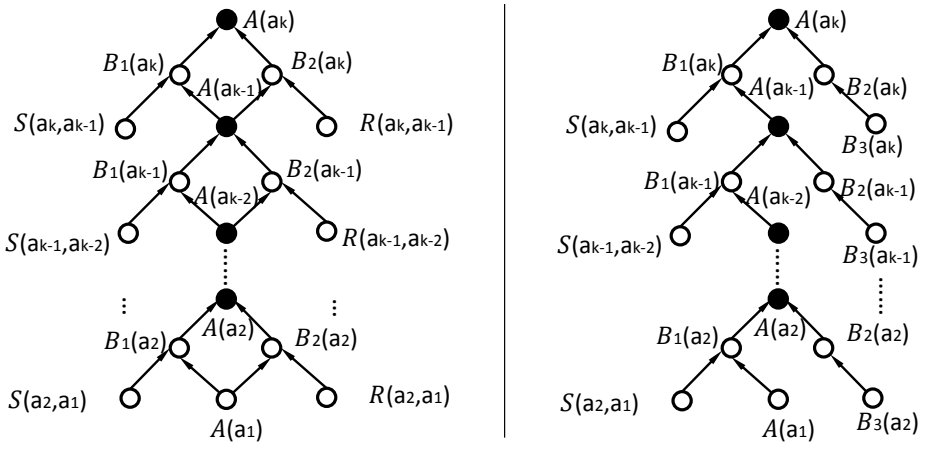

Figure 4: (left) The materialization graph of O_{ex_6}; (right) The materialization graph of O_{ex_7}.

Example 6. *Given a DHL ontology O_{ex_6} where its 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 is an integer greater than 2. We denote the corresponding datalog program of 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 $\mathcal{G}_{ex_6}$ is the unique materialization graph of P_{ex_6}. Observe that there exists a path (e.g., $A(a_1), B_1(a_2), A(a_2), ..., A(a_k)$) 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 $\mathcal{G}_{ex_6}$, and, $A(a_i)$ can also be added to $\mathcal{G}_{ex_6}$. Similar to the datalog program class $\mathbb{P}_{ex_1}$, we can also get a datalog program class $\mathbb{P}_{ex_6}$ 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 $\mathbb{P}_{ex_6}$ in a poly-logarithmical number of iterations. The intuitive reason is that, at least two paths exist starting from $A(a_1)$ to $A(a_k)$. These paths *twist* mutually and share the same joint nodes (see the black nodes). It makes the optimization of acceleration used in Algorithm A_{opt} invalid. 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 $\mathcal{G}_{ex_6}$, there would not exist an available SWD path for $A(a_i)$. We use '*path twisting*' to represent such cases.

In order to make Algorithm A_{opt} terminate in a poly-logarithmical 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 $\mathcal{O}_{ex_7}$ 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 is 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 $A(a_2), ..., A(a_k)$. Further, when constructing $\mathcal{G}_{ex_7}$, 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 $\mathcal{G}_{ex_7}$, since they have corresponding SWD paths; after that, all the other nodes can be added to $\mathcal{G}_{ex_7}$ 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 it *simple-concept restriction*). Intuitively, for the restricted DHL ontologies, the situation of 'path twisting' would not happen. This is because, if in each axiom of the form $B_1 \sqcap B_2 \sqsubseteq A$, w.l.o.g., B_1 is a simple concept, then none of 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-logarithmical 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 A^{ψ}_{opt} for some poly-logarithmical 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. Parallelly Tractable Materialization of DHL(∘)

In this part, we study parallelly tractable materialization of 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 its 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 is an integer greater than 2. The corresponding datalog program P_{ex_5} 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_5} constructed by Algorithm A_{opt} is denoted by $\mathcal{G}_{ex_9}$.*

One can check that the materialization graph $\mathcal{G}_{ex_9}$ has the same shape as that of $\mathcal{G}_{ex_6}$ in Figure 4. A twisted path exists in $\mathcal{G}_{ex_9}$ 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*.[5] We now consider such an axiom $R_1 \circ R_2 \sqsubseteq R$ (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 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'.[6] The reason 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. The similar analysis applies to such 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 *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 should be a* simple concept*;*
2. *for all axioms of the form $R_1 \circ R_2 \sqsubseteq R$, if there does not exits a transitive role S such that $R \sqsubseteq_* S$, then, at least one of R_1 and R_2 is a* simple role*; otherwise, both of R_1 and R_2 are* simple roles*.*

Example 10. *For the ontology O_{ex_7} in Example 9, all of the roles R_1, R_2 and R are non-simple roles. Thus, O_{ex_7} 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}.*

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

[6] 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.

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

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

5. A Further Optimized Algorithm for DHL(∘) Materialization

In this section, we first discuss that 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 follow both of the simple-concept and the role-concept restrictions.

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 by checking all possible rule instantiations in P^* where P is a datalog program that corresponds to an ontology. One can check that $|P^*|$ could be the square or cube of the number of constants occurring in P. Consider a datalog rule of the form '$R(x, y), A(y) \rightarrow B(x)$' that is rewritten from the axiom of the form $\exists R.A \sqsubseteq B$; since there are two variables in this rule, the number of all possible rule instantiations is $|\mathbf{IN}|^2$; if there are 1000 individuals in $\mathbf{IN}$, this number is 1 million. Similarly, for a datalog rule rewritten from an axiom of the form (R3) or (R4), the number of all possible rule instantiations is $|\mathbf{IN}|^3$. It is no doubt that a plain implementation of Algorithm A_{opt} would be delayed 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(∘) is our focus, we explain how to apply the above strategy in DHL(∘) 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 $|\mathbf{IN}|$. This is because that, 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 the datalog rules that have two body atoms, i.e., the datalog rules of the forms '$R(x, y), A(y) \rightarrow B(x)$' (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 $k|\mathbf{IN}|$ 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 the same 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 a `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 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 a `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 involved) 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 the step (**i**) of Algorithm OPT to Algorithm PRC. Algorithm A_{prc} is thus given as follows:

`Algorithm` A_{prc}. Given a DHL($\circ$) 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}. We next use an example to show how Algorithm A_{prc} handles DHL(∘) materialization.

Example 11. *Consider performing Algorithm A_{prc} on the ontology $\mathcal{O}_{ex_1}$ in Example 1. Note that the individual set $\boldsymbol{IN}$= $\{a_1, ..., a_k, b\}$ where $k + 1$ 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 $\forall o \in \boldsymbol{IN}$, 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 follow the simple-concept and the simple-role restrictions. The correctness of this theorem is based on 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. The detailed analysis can be found in the proof.

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 Algorithm A^{ψ}_{prc} outputs a materialization graph of $\mathcal{O}$.*

6. Evaluation

In the first part of this section, we analyze different kinds of ontologies and datasets and investigate the usability of the results proposed above. In the second part, we evaluate the implementation of Algorithm A_{prc} and compare it to the reasoning system RDFox.

6.1. Practical Usability of the Theoretical Results

In this part, we analyze different kinds of ontologies and datasets including benchmarks, real-world ontologies and datasets that can be expressed in ontology languages. Based on the analysis of these datasets, we find that, ignoring imports, many of them belong to $\mathcal{D}_{dhl}$ or $\mathcal{D}_{dhl(\circ)}$.

Benchmarks. In the Semantic Web community, many benchmarks are proposed to facilitate the evaluation of ontology-based systems in a standard and systematic way. We investigate several popular benchmarks using our results and find that the ontologies used in some benchmarks have simple structured TBoxes that can be expressed in RDFS and belong to $\mathcal{D}_{dhl}$. These benchmarks include SIB[7] (*Social Network Intelligence BenchMark*), BSBM[8] (*Berlin SPARQL*

[7]https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark
[8]http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

Benchmark) and LODIB[9] (*Linked Open Data Integration Benchmark*). The ontology used in IIMB[10] (*The ISLab Instance Matching Benchmark*) follows the simple-concept restriction.

In the latest version of LUBM[11] (The Lehigh University Benchmark), there are 48 classes and 32 properties. Statements about properties, such as inverse property statements, can be rewritten into datalog rules allowed in $\mathcal{D}_{dhl}$. Most of the statements about classes can be rewritten into datalog rules that are allowed in $\mathcal{D}_{dhl}$. Five axioms have, however, the form $A \sqsubseteq \exists R.B$, which requires existentially quantified variables in the rule head when rewriting the axiom into a logic rule: $A(x) \rightarrow \exists y(R(x, y) \wedge B(y))$, where a new anonymous constant y is introduced. This kind of rule is not considered when using OWL RL reasoners to handle LUBM [11, 25]. On the other hand, in some cases, this kind of rule can also be eliminated when taking a rewriting approach [26]. In summary, if the above kind of rule is not considered, the materialization of a LUBM dataset can be handled by Algorithm A^{ψ}_{opt}.

YAGO. The knowledge base YAGO[12] is constructed from Wikipedia and WordNet and the latest version YAGO3 [27] has more than 10 million entities (e.g., persons, organizations, cities, etc.) and contains more than 120 million facts about these entities. In order to balance the expressiveness and computing efficiency, a YAGO-style language, called YAGO *model*, is proposed based on a slight extension of RDFS [28]. In addition to the expressiveness of RDFS, YAGO *model* also allows stating the *transitivity* and *acyclicity* of a property. Making full use of RDFS features cannot lead to parallel tractability [19]. However, in [28], a group of materialization rules is specified, which is more efficient. All of these rules are allowed in $\mathcal{D}_{dhl}$. Thus, we have that a well-constructed YAGO dataset belongs to $\mathcal{D}_{dhl}$.

Real Ontologies. We investigated 151 ontologies that cover many domains like biomedicine, geography, etc. These ontologies are collected from the Protege ontology library,[13] Swoogle[14] and Oxford ontology lib.[15] All ontologies are available online.[16] Among these ontologies, 111 of them belong to $\mathcal{D}_{dhl}$ or $\mathcal{D}_{dhl(\circ)}$, and 21 DHL ontologies contain conjunctions and follow the simple-concept restriction. The remaining ontologies have simple TBoxes, i.e., no conjunction ($A_1 \sqcap A_2$) appears in these ontologies. We also find two DHL($\circ$) ontologies that follow the simple-role restriction.

The above investigation indicates that the simple-concept and the simple-role restrictions are allowed in many real applications of different fields. From the perspective of developers who work on building their own ontologies, they can also refer to $\mathcal{D}_{dhl}$ and $\mathcal{D}_{dhl(\circ)}$ to achieve the guarantee of the parallel tractability.

6.2. Evaluating the Implementation of Algorithm A_{prc}

We implemented a prototype system ParallelDHL for DHL($\circ$) materialization based on Algorithm A_{prc}. In this part, we evaluate ParallelDHL and compare it to the state-of-the-art reasoning system, RDFox [4], that can also handle ontology materialization.

[9] http://wifo5-03.informatik.uni-mannheim.de/bizer/lodib/
[10] http://islab.di.unimi.it/iimb/
[11] http://swat.cse.lehigh.edu/projects/lubm/
[12] http://www.mpi-inf.mpg.de/home/
[13] http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library
[14] http://swoogle.umbc.edu/
[15] http://www.cs.ox.ac.uk/isg/ontologies/lib/
[16] https://github.com/quanzz/PT

Datasets. We select eight ontologies from the data sources given in the previous subsection; four of the eight ontologies belong to $\mathcal{D}_{dhl(\circ)}$ and the rest of them do not follow the simple-concept or the simple-role restriction. The eight ontologies are real ontologies and are applied in different fields. The basic information of these ontologies is summarized in Table 2 (we use $\mathcal{D}^{-}_{dhl(\circ)}$ to denote the complementary set of $\mathcal{D}_{dhl(\circ)}$).

Table 2: The Information of the Test Ontologies

Ontology Name	Field	$\lvert\mathcal{T}\rvert + \lvert\mathcal{R}\rvert^a$	$\lvert\mathcal{A}\rvert^b$	Type^c
Finance	finance	1,934	6,152	$\mathcal{D}^{-}_{dhl(\circ)}$
Molecule	chemistry	43	0	$\mathcal{D}^{-}_{dhl(\circ)}$
GrossAnatomy	anatomy	2,276	13	$\mathcal{D}^{-}_{dhl(\circ)}$
Skeleton	medicine	815	0	$\mathcal{D}^{-}_{dhl(\circ)}$
ChemistryPrimitive	chemistry	167	0	$\mathcal{D}_{dhl(\circ)}$
FacebookOnto	social networking	185	28	$\mathcal{D}_{dhl(\circ)}$
Mahabharata	literature	69	2,036	$\mathcal{D}_{dhl(\circ)}$
Transportation	traffic	925	511	$\mathcal{D}_{dhl(\circ)}$

[a] $\lvert\mathcal{T}\rvert + \lvert\mathcal{R}\rvert$ denotes the number of axioms occurring in the ontology.

[b] $\lvert\mathcal{A}\rvert$ denotes the number of assertions occurring in the ABox.

[c] $\mathcal{D}_{dhl(\circ)}$ (resp., $\mathcal{D}^{-}_{dhl(\circ)}$) means that the ontology belongs to $\mathcal{D}_{dhl(\circ)}$ (resp., $\mathcal{D}^{-}_{dhl(\circ)}$).

For the purpose of comparability in the ABox scale, we consider generating ABox assertions to a limited number with respect to the TBoxes and RBoxes of the test ontologies. The generation method is based on the work [29] where a benchmark generator is proposed for evaluating ontology-based systems. For each test ontology, e.g., the ontology Finance, we generate 5 new ontologies with different ABoxes; these ontologies are denoted by Finance-i ($i \in \{1, 2, 3, 4, 5\}$) where i represents that the number of assertions in the ABox of Finance-i reaches to $i \times 100,000$; we also use "Finance series" to denote the five generated ontologies for the ontology Finance. The other test ontologies are processed similarly.

The Experimental Results. We run ParallelDHL and RDFox over the above eight ontology series respectively. The running environment is a DELL server with a memory of 16 GiB and 4 cores. For fairness, we set the same number of threads (i.e., 1,2,4,6 and 8 threads respectively) to ParallelDHL and RDFox in each experiment. The results of reasoning times[17] are presented in the 16 line graphs (denoted by $lg1$,...,$lg16$) of Figure 5, where the abscissa of each line graph records the numbers of ABox assertions, the ordinate records the values of reasoning time (millisecond per unit), and each of the five curves in different colors denotes the trend of reasoning time with the corresponding number of threads allocated (we use line-k to denote the curve corresponding to k threads).

We further process the collected data and fill the results in Table 3, where each cell corresponds to a test ontology (see the row label) and a new generated ABox (distinguished by the column labels); the three values from above to below in each cell are: (1) *the minimal time ratio* - the minimal reasoning time of ParallelDHL divided by the minimal reasoning time of RDFox; (2) *the average speedup of ParallelDHL*;[18] (3) *the average speedup of RDFox*. The minimal time ratio describes the performance of ParallelDHL by using RDFox as the baseline. The indicator

[17] The experimental results can be found at https://github.com/quanzz/PT.

[18] Suppose T_i is the reasoning time with i threads allocated, the average speedup is $\frac{1}{4}(\frac{T_1}{T_2} + \frac{T_2}{T_4} + \frac{T_4}{T_6} + \frac{T_6}{T_8})$.

speedup and its derived indicators are the most common tools used for measuring the capacity of parallelism [4, 30, 11]. Here, we use the average speedup [31] to describe the average capacity of parallelism with different threads allocated. In the following, we give the detailed analysis based on the contents in Figure 5 and Table 3.

The Analysis of the Experimental Results. According to the theoretical results in Section 4, the ontologies belonging to $\mathcal{D}^{-}_{dhl(\circ)}$ may not be parallelly tractable in terms of materialization. In other words, parallel techniques may not work for improving the efficiency of materialization. This is shown in the line graphs of Figure 5. We can see that, in the line graphs ($lg1$-$lg8$) of the ontology series that belong to $\mathcal{D}^{-}_{dhl(\circ)}$, the four lines, line-2, line-4, line-6 and line-8, intersect to some degree. This situation in the line graphs for Finance and GrossAnatomy series is more obvious. For example, in line graph $lg1$, line-2 stays higher than line-6, but lower than line-4. The intersection of lines indicates that reasoning time cannot be obviously reduced with more threads allocated. This is also supported by the results of the average speedups. From Table 3, we can see that the average speedups of ParallelDHL and RDFox for ChemistryPrimitive, FacebookOnto, Mahabharata and Transportation series are more stable compared with the other ones, i.e., they are averagely 1.4 (it means that the reasoning time can be reduced averagely by 1.4 times with two more threads allocated) for ParallelDHL, and, 1.1 for RDFox. For the four ontology series that belong to $\mathcal{D}^{-}_{dhl(\circ)}$, the average speedups have a higher volatility. For example, Finance and Skeleton series lead to high average speedups that reach up to 3, while, for Molecule and GrossAnatomy series, the average speedups are even lower than 1. In summary, from the experiments on the test ontology series, parallelism leads to a more effective improvement for materializing the ontology series that belong to $\mathcal{D}_{dhl(\circ)}$ compared with the ones in $\mathcal{D}^{-}_{dhl(\circ)}$.

By analyzing the experimental results of ParallelDHL and RDFox, we have that ParallelDHL is a competitive system. From Table 3, we can find that most of the minimal time ratios are close to 1. This means that the reasoning times of ParallelDHL are close to that of RDFox in parallel. In particular, for GrossAnatomy, FacebookOnto and Transportation series, ParallelDHL performs much better than RDFox; therein, for materializing FacebookOnto-3, ParallelDHL only costs one fifth of the minimal reasoning time of RDFox. On the other hand, for most ontology series, the average speedups of ParallelDHL are overall higher than that of RDFox. The main difference of ParallelDHL and RDFox lies in that ParallelDHL applies the optimizations designed for Algorithm A_{opt} (see Section 3). The higher average speedups of PrallelDHL also verifies the validity of the optimizations used in Algorithm A_{opt}. One may note that, ParallelDHL is averagely 3 times slower than RDFox when handling Skeleton series. The main reason is that the computation of `rch` relations occupies a large amount of time. We have checked that, the situation of path twisting occurs in Skeleton series. This makes the the optimization based on `rch` relations invalid as discussed in Section 4.

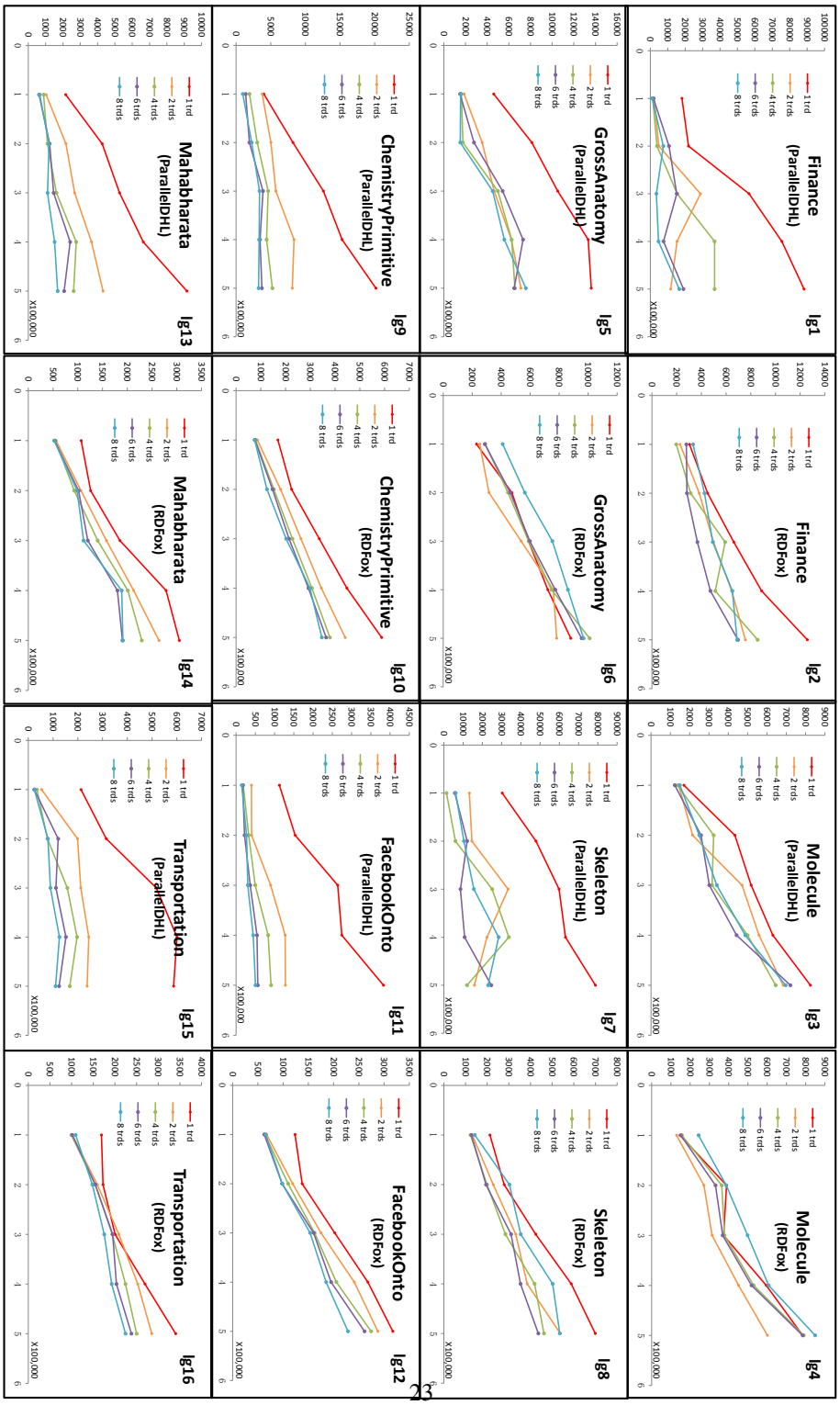

Figure 5: The line graphs of all experiments. The 16 line graphs are labeled by $lg1,...,lg16$ respectively.

Table 3: The Analysis of Reasoning Results

	1	2	3	4	5
	0.54	1.36	0.94	0.99	1.70
Finance	3.06	1.88	2.29	2.95	2.73
	1.01	1.04	1.13	1.11	1.19
	0.94	0.79	0.96	0.98	1.07
Molecule	1.04	1.24	1.13	1.07	1.05
	0.90	1.03	0.95	1.01	0.99
	0.65	0.48	0.84	0.77	0.83
GrossAnatomy	1.41	1.69	1.33	1.32	1.22
	0.87	0.99	0.94	0.96	0.98
	1.11	**3.08**	**3.03**	**3.04**	**2.75**
Skeleton	3.28	1.84	1.65	1.74	1.96
	1.13	1.01	1.06	1.08	1.08
	1.29	1.54	1.68	1.14	0.93
ChemistryPrimitive	**1.46**	**1.43**	**1.44**	**1.51**	**1.65**
	1.28	1.16	1.14	1.11	1.15
	0.23	**0.22**	**0.19**	**0.24**	**0.22**
FacebookOnto	**1.81**	**1.83**	**1.82**	**1.61**	**1.78**
	1.22	1.08	1.07	1.09	1.09
	1.19	1.19	0.99	0.84	0.89
Mahabharata	**1.41**	**1.46**	**1.51**	**1.47**	**1.56**
	1.24	1.07	1.13	1.10	1.12
	0.24	0.52	0.51	0.65	0.49
Transportation	**1.98**	**1.57**	**1.61**	**1.55**	**1.59**
	1.14	1.04	1.03	1.08	1.11

7. Related Work

The parallel reasoner RDFox [4] is used to evaluate our implementation. RDFox is a state-of-the-art system that handles reasoning on 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 difference lies in that, a group of rule instantiations are handled by one processor (namely a thread) in RDFox, while in Algorithm A_{bsc}, each rule instantiation is assigned to a unique processor. Thus, in theory, the materialization of the datalog program in Example 1 is serial on RDFox. It is also shown from the experiments that parallelism does not lead to a remarkable improvement when RDFox handles the ontology series in $\mathcal{D}^{-}_{dhl(\circ)}$. We use Algorithm A_{opt} to show that the ontology in Example 1 is also parallelly tractable. Our implementation ParallelDHL, which is based on Algorithm A_{prc}, has also been shown to have a better performance than RDFox when handling the test ontologies that belong to $\mathcal{D}_{dhl(\circ)}$.

There is work that studies parallel reasoning in RDFS and OWL. The current methods mainly focus on optimizing reasoning algorithms from different aspects. The authors of [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 the parallel RDF reasoning and query answering. In the work of [5], the RETE algorithm is used to accelerate rule matching for RDFS reasoning. The authors of [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 [32, 25]. 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 [30], OWL QL [33], and even highly expressive languages [7, 34, 35, 8]. In current work, several techniques are proposed to adapt parallel computation to OWL reasoning tasks. A kind of graph-based method is discussed in [33] to enhance OWL QL classification. The authors of [34, 35, 8] propose pruning techniques to optimize the Tableaux algorithm. The *lock-free technique* is applied in the work [30, 7].

Another line of optimizing parallel reasoning is to utilize high-performance computing platforms. For in-memory platforms, different supercomputers, like Cray XMT, Yahoo S4, have early been used in parallel RDF reasoning [9, 10]. The authors of [12] report their work on RDF-S reasoning based on massively parallel GPU hardware. The distributed parallel platforms, like MapReduce and Peer-to-Peer networks, are also used for RDFS reasoning. The representative systems are WebPIE [11], Marvin [36] and SAOR [37]. 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, we have to discuss more issues, 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 the datalog rewritable ontologies. To identify the parallelly tractable classes, we proposed two NC algorithms, Algorithm A^{ψ}_{bsc} and Algorithm A^{ψ}_{opt}, that perform materialization on datalog rewritable ontology languages. Based on these algorithms, we identified the corresponding parallelly tractable datalog program (PTD) classes such that materialization on the datalog programs in these classes is in the complexity class NC. 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 parallelly tractable. For DHL and DHL($\circ$), we proposed two restrictions such that materialization is parallelly tractable.

We verified the usefulness of our theoretical techniques in two ways. On the one hand, we analyzed different kinds of datasets, including well-known benchmarks, real-world ontologies and a famous dataset YAGO. Our analysis showed that many real ontologies belong to the parallelly tractable class $\mathcal{D}_{dhl}$ or $\mathcal{D}_{dhl(\circ)}$. The developers and users can also refer to $\mathcal{D}_{dhl}$ and $\mathcal{D}_{dhl(\circ)}$ to create large-scale ontologies for which parallel tractability is theoretically guaranteed. 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, and can be implemented for practice. 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 optimization proposed in this paper results in a better performance on parallelly tractable ontologies compared with RDFox.

In the future work, we will extend our work in two lines. One line is a further study of the parallel tractability of other expressive ontology languages, in addition to datalog rewritable ones.

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 parallelly tractable. Another line of the future 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, like ontology classification, ontology debugging and query answering.

Acknowledgments

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

Reference

[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, 2016, pp. 75–76.
[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, 2015, pp. 15:1–15:11.
[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, 2014, pp. 129–137.
[5] M. Peters, S. Sachweh, A. Zündorf, Large scale rule-based reasoning using a laptop, in: Proc. of ESWC, 2015, pp. 104–118.
[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, 2012, pp. 675–690.
[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, 2011, pp. 31–45.
[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, 2012, pp. 133–148.
[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, 2010, pp. 436–452.
[15] B. N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: combining logic programs with description logic, in: Proc. of WWW, 2003, pp. 48–57.
[16] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[17] M. Krötzsch, S. Rudolph, P. Hitzler, Complexity boundaries for horn description logics, in: Proc. of AAAI, 2007, pp. 452–457.
[18] 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, 2009, pp. 2040–2045.
[19] 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.
[20] 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.
[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, 2010, pp. 938–948.
[24] E. Allender, Reachability problems: An update, in: Proc. of CiE, 2007, pp. 25–27.

[25] J. Weaver, J. A. Hendler, Parallel materialization of the finite RDFS closure for hundreds of millions of triples, in: Proc. of ISWC, 2009, pp. 682–697.
[26] B. C. 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] F. Mahdisoltani, J. Biega, F. M. Suchanek, YAGO3: A knowledge base from multilingual wikipedias, in: Proc. of CIDR, 2015.
[28] F. M. Suchanek, G. Kasneci, G. Weikum, YAGO: A large ontology from wikipedia and wordnet, J. Web Sem. 6 (3) (2008) 203–217.
[29] Q. Elhaik, M.-C. Rousset, B. Ycart, Generating random benchmarks for description logics, in: Proc. of DL, 1998, pp. 231–235.
[30] 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.
[31] N. Ichiyoshi, K. Kimura, Asymptotic load balance of distributed hash tables, in: Proc. of FGCS, 1992, pp. 869–876.
[32] R. Soma, V. K. Prasanna, A data partitioning approach for parallelizing rule based inferencing for materialized OWL knowledge bases, in: Proc. of ISCA, 2008, pp. 19–25.
[33] D. Lembo, V. Santarelli, D. F. Savo, A graph-based approach for classifying OWL 2 QL ontologies, in: Proc. of DL, 2013, pp. 747–759.
[34] T. Liebig, F. Müller, Parallelizing tableaux-based description logic reasoning, in: Proc. of OTM Workshops, 2007, pp. 1135–1144.
[35] A. Schlicht, H. Stuckenschmidt, Distributed resolution for ALC, in: Proc. of DL, 2008, pp. 326–341.
[36] 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.
[37] 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 equals 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, $P \in \mathcal{D}_{\mathsf{A}_{bsc}^{\psi}}$ iff P has a materialization graph whose depth is upper-bounded by $\psi(|P|)$.*

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 explicit node v, $\texttt{depth}(v)$=0; for each implicit node v' whose parents are $v_1, ..., v_i$, $\texttt{depth}(v')$=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 $\texttt{depth}(\mathcal{G})$.

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

($\Leftarrow$) Suppose P has a materialization graph $\mathcal{G}$ whose depth is upper-bounded by $\psi(|P|)$. 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 $\psi(|P|)$. 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 the output $\mathcal{G}$ is a materialization graph of P.*

Proof: this lemma can be proved by 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 explicit 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 the 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 $\mathcal{G}^i$ in 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 $e(B_1, H), ..., e(B_n, H)$ are created. (2) when 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 $e(B_1, H), ..., e(B_n, H)$ 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 $e(B_1, H), ..., e(B_n, H)$ 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 in the proof for 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 the length 1. This is because that 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 *Given a DL-Lite ontology $\mathcal{O}$, for any materialization graph $\mathcal{G}$ of $\mathcal{O}$, each atom of the form $A(x)$ ($A \neq \bot$) or $R(x, y)$ in $\mathcal{G}$ has an SWD path.*

Proof: this lemma can be proved by an induction on applications of the datalog rules corresponding to DL-Lite axioms.

Base case. For each explicit 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 '$B_1, ..., B_n \rightarrow H$' that is rewritten from some DL-Lite axiom, we have the induction hypothesis that $B_1, ..., B_n$ 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 distinguish 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 $(v_1, ..., v_n, B(x))$. We have that, in some materialization graph, node $A(x)$ has an SWD path of the form $(v_1, ..., v_n, B(x), A(x))$.

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

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

In both of 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 $\mathcal{O}$. □

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 ontology $\mathcal{O}$, there always exists a poly-logarithmically bounded function ψ such that A^{ψ}_{opt} can handle the materialization of $\mathcal{O}$, since in any materialization graph, each node has an SWD path. More precisely, we can set 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, a rule with a binary atom as its head can only be either of the form (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 $\mathcal{A}^{\psi}_{opt}$ for some poly-logarithmical function ψ, then the whole materialization can be handled by $\mathcal{A}^{\psi}_{opt}$. 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 computation of transitive roles. As mentioned before, there exists an NC algorithm for transitivity computation. Inspired by this, we can prove that $\mathcal{A}^{\psi}_{opt}$ *handles Stage 1 for some poly-logarithmical function* ψ. The proof of this result can be shown by separately considering non-transitive roles and the roles that are transitive or influenced by transitive roles. Specifically, we say that role R is *transitively influenced* (TI) if (1) $R \circ R \sqsubseteq R \in \mathcal{R}$; or (2) there exists a TI role R' such that $R' \sqsubseteq_* R$ or $R' \sqsubseteq_* R^-$. We say that a role is an NTI (non-transitively influenced) role if it is not a TI role. We further define a set δ_R for each role $R \in \mathbf{R}$ as follows:

Definition 6. *For each $R \in \boldsymbol{R}$, let δ_R be the set of all assertions as follows:*

1. *for each $R(a, b) \in \mathcal{A}$;*

2. $R(a, b)$, *for each* $R'(a, b) \in \mathcal{A}$ *and* $R' \sqsubseteq_* R$;

3. $R(a, b)$, *for each* $R'(b, a) \in \mathcal{A}$ *and* $R' \sqsubseteq_* R^-$.

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

Lemma 4. *$P \models R(a, b)$ implies: (1) if R is an NTI 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 TI 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 an NTI 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 TI 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 $\mathcal{A}_{opt}^{\psi}$. Further, transitive computation in part (2) can also be handled by $\mathcal{A}_{opt}^{\psi}$. Thus, there exists a poly-logarithmical function ψ such that $\mathcal{A}_{opt}^{\psi}$ handles Stage 1.

The results of Stage 1 serve as the facts of Stage 2. In other words, all binary atoms are explicit nodes in Stage 2. This also means that the rules of the forms (T1) and (T3) generate nodes with SWD paths. Due to the simple-concept restriction, for each rule instantiation $A_1(a), A_2(a) \rightarrow B(a)$, one of $A_1(a)$ and $A_2(a)$ always has an SWD path. Thus, Stage 2 can be handled by $\mathcal{A}_{opt}$ in at most two iterations. 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 $R(a, b)$.

Basic case. If $R(a, b) \in \mathbf{I}$, $R(a, b) \in \delta_R$. In this case, regardless of that R is a TI or an NTI role, all of (1), (2) and (3) hold.

Inductive case. We first study the case where R is an NTI 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 an NTI role, R' has to be an NTI role according to Definition 6. 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 6; (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 $R'' \sqsubseteq R'^-$, this case is similar to (Case 1.1.2).

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 an NTI 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 an NTI role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$. Similar 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 TI 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 $\delta_{R''}^*$ is the transitive closure of $\delta_{R''}$, we then have that there must exist such atoms $(R''(a,c_1), R''(c_1,c_2), ..., R''(c_n,b))$ in $\delta_{R''}$. Further, due to $R'' \sqsubseteq_* R$ and Definition 6, we have $R(a,c_1), R(c_1,c_2), ..., R(c_n,b) \in \delta_R$. 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 an NTI role. By induction hypothesis, $R'(b,a) \in \delta_{R'}$. Similar 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 TI 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 $(R''(b,c_1), R''(c_1,c_2), ..., R''(c_n,a))$ in $\delta_{R''}$. Further, due to $R'' \sqsubseteq_* R^-$ and Definition 6, we have $R(a,c_n), R(c_n,c_{n-1}), ..., R(c_1,b) \in \delta_R$. 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 TI 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 an NTI role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$ holds. Similar 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 TI 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 DHL. Thus we only consider Stage 1 here.

Our target is to show that $\mathcal{A}^{\psi}_{opt}$ handles Stage 1. To this end, we also distinguish all roles by whether they are transitively influenced. Since we have to consider complex RIAs, we re-define TI and NTI roles as follows: a role R is *transitively influenced* (TI) if (1) $R \circ R \sqsubseteq R \in \mathcal{R}$; or (2) there exists a TI role R' such that $R' \sqsubseteq_* R$ or $R' \sqsubseteq_* R^-$; or (3) there exist a TI role R' and an axiom of either of the form $R' \circ R'' \sqsubseteq R$ or $R'' \circ R' \sqsubseteq R$. We say that a role is an NTI role if it is not transitively influenced. The set δ_R for each role $R \in \mathbf{R}$ is re-defined based on *role sequence set* that is defined as follows:

Definition 7. *Let $\mathcal{L}(R)$ be the set of role sequences with respect to R as follows:*

1. *R', for each $R' \sqsubseteq_* R$;*
2. *R'^-, for each $R' \sqsubseteq_* R^-$;*
3. *L_1L_2, for each axiom of the form (except R3) $R_1 \circ R_2 \sqsubseteq R' \in \mathcal{R}$, $L_i \in \mathcal{L}(R_i)(1 \leq i \leq 2)$ and $R' \in \mathcal{L}(R)$;*
4. *$L_2^-L_1^-$, for each axiom of the form (except R3) $R_1 \circ R_2 \sqsubseteq R' \in \mathcal{R}$, $L_i \in \mathcal{L}(R_i)(1 \leq i \leq 2)$ and $R'^- \in \mathcal{L}(R)$.*

In the above definition, for a role sequence $L = R_1R_2, ..., R_n$, let $L^- = R_n^-, ..., R_2^-R_1^-$. We then give the following definition for δ_R.

Definition 8. *For each $R \in \boldsymbol{RN}$, let δ_R be the set of all assertions as follows:*

1. *for each $R(a, b) \in \mathcal{A}$;*
2. *$R(a, b)$, for each $R'(a, b) \in \mathcal{A}$ and $R' \sqsubseteq_* R$;*
3. *$R(a, b)$, for each $R'(b, a) \in \mathcal{A}$ and $R' \sqsubseteq_* R^-$;*
4. *$R(a, b)$, for each $R_0R_2, ..., R_n \in \mathcal{L}(R)$, where $R_i(x_i, x_{i+1}) \in \mathcal{A}$ and $x_0 = a, x_{n+1} = b$ for $0 \leq i \leq n$.*

Lemma 5. *$P \models R(a, b)$ implies: (1) if R is an NTI 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 TI but not transitive role, then there exists a role sequence $R_1R_2, ..., R_n \in \mathcal{L}(R)$ 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 $1 \leq i \leq n$ 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 an NTI 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 TI 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 in the left hand side of (R4) are restricted to be a simple role. Thus, the computations of (1) and (3) can be handled by $\mathcal{A}^{\psi}_{opt}$. Further, transitive computation in part (2) can also be handled by $\mathcal{A}^{\psi}_{opt}$. In summary, there exists a poly-logarithmical function ψ such that $\mathcal{A}^{\psi}_{opt}$ 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 $R(a, b)$.

Basic case. If $R(a, b) \in \mathbf{I}$, $R(a, b) \in \delta_R$. In this case, regardless of the case that R is a TI or an NTI role, all of (1), (2) and (3) hold.

Inductive case. We first study the case where R is an NTI 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 an NTI role, R' has to be an NTI role according to Definition 8. 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 8; (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 $R'' \sqsubseteq R'^-$, this case is similar to (Case 1.1.2); (Case 1.1.4) There exists $R_0R_2, ..., R_n \in \mathcal{L}(R')$, where $R_i(x_i, x_{i+1}) \in \mathcal{A}$ and $x_0 = a, x_{n+1} = b$ for $0 \leq i \leq n$. According to Definition 7, $R_0R_2, ..., R_n \in \mathcal{L}(R)$ holds since $R' \sqsubseteq R \in \mathcal{R}$.

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 an NTI role, R' has to be an NTI role according to Definition 8. 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 8; (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 $R_0R_2, ..., R_n \in \mathcal{L}(R')$, where $R_i(x_i, x_{i+1}) \in \mathcal{A}$ and $x_0 = b, x_{n+1} = a$ for $0 \leq i \leq n$. According to Definition 7, $R_n^-, ..., R_1^- \in \mathcal{L}(R)$ holds since $R' \sqsubseteq R^- \in \mathcal{R}$.

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 an NTI role, both of R_1 and R_2 are NTI roles. By the induction hypothesis, for each $R_i(c_i, c_{i+1})(1 \leq i \leq 2)$ 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 in following different 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 $R_0R_2, ..., R_m \in \mathcal{L}(R')$, where $R_j(x_j, x_{j+1}) \in \mathcal{A}$ and $x_0 = c_i, x_{n+1} = c_{i+1}$ for $0 \leq i \leq m$ and $R' \sqsubseteq_* R_i$, then $L_i = R_0R_2, ..., R_m$ holds.

Based on $L_i(1 \leq i \leq 2)$, 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 an NTI but not 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 an NTI role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$. Similar 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 TI but not a transitive role. By the induction hypothesis, there exists $R_1, ..., R_n \in \mathcal{L}(R')$ 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 $1 \leq i \leq n$ and $x_0 = a, x_{n+1} = b$. Since $R' \sqsubseteq R$ holds, $R_1, ..., R_n \in \mathcal{L}(R)$ 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 $R'(a,c_1), R'(c_1,c_2), ..., R'(c_n,b) \in \delta_{R'}$. Since $R' \sqsubseteq R \in \mathcal{R}$ holds, we also have that $R(a,c_1), R(c_1,c_2)\ , ..., R(c_n,b) \in \delta_R$. 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 an NTI role. By the induction hypothesis, $R'(b,a) \in \delta_{R'}$. Similar 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 TI but not transitive role. By the induction hypothesis, there exists $R_1, ..., R_n \in \mathcal{L}(R')$ 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 $1 \leq i \leq n$ and $x_0 = b, x_{n+1} = a$. Since $R' \sqsubseteq R^-$ holds, $R_n^-, ..., R_1^- \in \mathcal{L}(R)$ 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 $R'(b,c_1), R'(c_1,c_2), ..., R'(c_n,a) \in \delta_{R'}$. Since $R' \sqsubseteq R^- \in \mathcal{R}$, we also have that $R(a,c_1), R(c_1,c_2)\ , ..., R(c_n,b) \in \delta_R$. 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 NTI 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 TI 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 TI 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 an NTI role. By the induction hypothesis, $R'(a,b) \in \delta_{R'}$ holds. Similar 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 TI but not a transitive role. By the induction hypothesis, there exists $R_1, ..., R_n \in \mathcal{L}(R')$ 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 $1 \leq i \leq n$ and $x_0 = a, x_{n+1} = b$. Since $R' \sqsubseteq R$, $R_1, ..., R_n \in \mathcal{L}(R)$ 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 7, $R' \in \mathcal{L}(R)$ 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 an NTI role. By the induction hypothesis, $R'(b,a) \in \delta_{R'}$. Similar 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 TI but not a transitive role. By the induction hypothesis, there exists $R_1, ..., R_n \in \mathcal{L}(R')$ 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 $1 \leq i \leq n$ and $x_0 = b, x_{n+1} = a$. Since $R' \sqsubseteq R^-$, $R_n^-, ..., R_1^- \in \mathcal{L}(R)$ 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 TI 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 an NTI 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 $R_1 L_{R_2} \in \mathcal{L}(R)$ which 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: similar 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: similar to what we do when proving Theorem 3 and Theorem 4, we separate the materialization of a DHL(∘) ontology into two stages: the role materialization (all the rules of the forms (R1-R4) are exhaustively applied) and the concept materialization (the rules of the forms (T1-T3) are then applied). The difference is that, we further separate the role materialization into successively two stages: *the simple role materialization* (Stage SRM) and *the non-simple role materialization* (Stage NSRM). Similarly, the concept materialization is also separated into successively two 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 in Stage SCM, 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 $\mathtt{rch}(B(a), A(a))$ 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 $\mathtt{rch}(B(b), A(a))$ 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 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. The case of the rule instantiation of the form $B(a) \rightarrow A(a)$ (resp., $R(a,b), B(b) \rightarrow A(a)$) is referred to Case 1 (resp., Case 2). 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 $\mathtt{rch}(A_2(a), A(a))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of the SWD path for $A_2(a)$ can be determined. Thus, the existence of the SWD path of $A(a)$ can also be determined. It is similar for the case where $A_2(a)$ is derived and A_2 is a simple concept.

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

We now discuss Stage NSRM. For each atom of the form $R(a,b)$ where R is not a simple role and 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 $\mathtt{rch}(S(a,b), R(a,b))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of the SWD path for $S(a,b)$ can be determined. Thus, the existence of the 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 $\mathtt{rch}(S(b,a), R(a,b))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of the SWD path for $S(b,a)$ can be determined. Thus, the existence of the 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 $R_1 \circ R_2 \sqsubseteq \mathcal{RR}$. If $R_1(a,c)$ is derived where R_1 is a simple role, we have that $\mathtt{rch}(R_2(c,b), R(a,b))$ is in S_{rch} according to Algorithm Prc. By the induction hypothesis, the existence of the SWD path for $R_2(c,b)$ can be determined. Thus, the existence of the SWD path of $R(a,b)$ can also be determined. It is similar for the case where $R_2(c,b)$ is derived and R_2 is a simple role.

We finally consider the derivation of the atoms of the form $R(a,b)$, where R is a transitive role. Recall 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 exists such axioms of the form $S_1 \circ S_2 \sqsubseteq S$ where S_1 and S_2 are simple roles. One can check the set δ_R can computed by A_{prc} in at most two iterations (see Case 4, Case 5 and Case 6). We should 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 hight of t is up-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. □