Exploring Parallel Tractability of Ontology Materialization

Zhangquan Zhou[1] and **Guilin Qi**[1] and **Birte Glimm**[2]

Abstract. Materialization is an important reasoning service for applications built on the Web Ontology Language (OWL). To make materialization efficient in practice, current research focuses on deciding tractability of an ontology language and designing parallel reasoning algorithms. However, 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, i.e., the reasoning is inherently sequential in the worst case. 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. In this work, we focus on datalog rewritable ontology languages. We identify several classes of datalog rewritable ontologies (called *parallelly tractable classes*) such that materialization over them is parallelly tractable. We further investigate the parallel tractability of materialization of a datalog rewritable OWL fragment DHL (*Description Horn Logic*) and an extension of DHL that allows *complex role inclusion axioms*. Based on 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.

1 Introduction

The Web Ontology Language OWL[3] is an important standard for ontology languages in the Semantic Web and other application areas. *Materialization* is a basic reasoning service for computing all implicit facts that follow from a given OWL ontology. Since there is an exponential growth of semantic data [18], it is challenging to perform materialization on such large-scale ontologies efficiently.

To make materialization sufficiently efficient and scalable in practice, many works employ parallel reasoning systems. For example, RDFox [19] is a parallel implementation for materialization of datalog rewritable ontology languages. Parallel reasoning is also studied for the ontology language RDFS [22, 26]. There are also parallel implementations for scalable reasoning of highly expressive ontology languages [25, 33]. However, according to [5], even for RDFS and datalog rewritable ontology languages, which have PTime-complete or higher complexity[4] of reasoning in the worst case, they are not parallelly tractable, i.e., reasoning may be inherently sequential even on a parallel implementation. On the other hand, some well-known large-scale ontologies, such as YAGO, have been shown to have good performance for parallel reasoning [14], but they are expressed in ontology languages that are not parallelly tractable. The theoretical results on the complexity of ontology languages 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, the aim of our study is to identify properties that make an ontology parallelly tractable and that can also guide ontology engineers in creating ontologies for which parallel tractability can be guaranteed theoretically. According to [19], 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. Our aim 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 [5]. 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 [5]. However, current materialization algorithms (e.g., the algorithm used in RDFox [19]) are not NC algorithms, since their computational complexity is PTime-complete. Thus, we study the parallel tractability of materialization by first giving several NC algorithms that perform materialization, and then identifying the corresponding classes of datalog rewritable ontologies (called *parallelly tractable classes*) that can be handled by these NC algorithms. We next study the specific ontology language *Description Horn Logic* (DHL) [6], which is a datalog rewritable fragment of OWL, and investigate what kinds of ontologies expressed in DHL are in the parallelly tractable classes. 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 ontologies following the proposed restrictions belong to the parallelly tractable classes.

The rest of the paper is organized as follows. In Section 2, we introduce some basic notions. We then give some NC algorithms in Section 3 and Section 4. We study the parallelly tractable materialization of DHL and its extension in Section 5 and Section 6 respectively. In Section 7, we analyze real-world datasets. We then discuss related work in Section 8 and conclude in Section 9. The technical report can be found at "https://github.com/quanzz/ECAI2016".

2 Preliminaries

In this section, we introduce some notions that are used in this paper.

[1] School of Computer Science and Engineering, Southeast University, email: {qzz, gqi}@seu.edu.cn

[2] Institution of Artificial Intelligence, University of Ulm, email: birte.glimm@uni-ulm.de

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

[4] We consider the data complexity for materialization here.

Datalog. We discuss the main issues in this paper using standard datalog notions. In datalog [1], 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: '$B_1, ..., B_n \rightarrow H$',[5] 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*.

DHL. DHL (short for *description horn logic*) [6] is introduced as an intersection of description logic (DL) and datalog in terms of expressivity. In what follows, **CN**, **RN** and **IN** 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 [12]. 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) and (T2); $\mathcal{R}$ 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-T2 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 $\mathbf{R}$. An axiom of the form $A \sqsubseteq B$ is a special case of (T1) where only one concept appears on the left-hand side. For an axiom of the form $A \sqsubseteq \forall R.B$ that is also allowed in DHL, we only consider its equivalent form $\exists R^-.A \sqsubseteq B$.

Table 1: Axioms and corresponding datalog rules

	Axioms	Datalog Rules
(T1)	$A_1 \sqcap A_2 \sqsubseteq B$	$A_1(x), A_2(x) \rightarrow B(x)$
(T2)	$\exists R.A \sqsubseteq B$	$R(x, y), A(y) \rightarrow B(x)$
(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)	$R(a, b)$	$R(a, b)$

In the initial work of DHL [6], *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).

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

A DHL (or DHL($\circ$)) ontology can be transformed to a datalog program (see the corresponding rules in the right column of Table 1). In what follows, for an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, 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 [10]. 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.

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 [6]. Reasoning with RDFS ontologies is NP-complete [29] and, thus, is not parallelly tractable. However, by applying some simplifications and restrictions, RDFS statements can be expressed in DHL axioms [6].

Ontology Materialization. Based on the above representations, ontology materialization corresponds to the evaluation of datalog programs. Specifically, given a datalog program $\langle R, \mathbf{I} \rangle$, 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 $T_R^\omega(\mathbf{I})$. In this paper, we consider the data complexity of materialization, i.e., we *assume that the rule set R is fixed*.

NC. The parallel complexity class NC, known as Nick′s Class [5], 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 parallel 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 [5].

From the perspective of implementations, the NC problems are also highly parallel feasible for other parallel models like BSP [31] and MapReduce [11]. 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 8) 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 Parallelly Tractable Class

Parallelly Tractable Class. Our target is to find for which kinds of ontologies (not ontology languages) materialization is parallelly tractable. Since we assume that for any datalog program $\langle R, \mathbf{I} \rangle$ the rule set R is fixed, the materialization problem is thus in data complexity PTime-complete, which is considered to be inherently sequential in the worst case [5]. 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 materializa-

tion 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). Formally, we give the following definition to identify such a class of datalog programs.

Definition 1. ***(Parallelly Tractable Class)*** *Given a class $\mathcal{D}$ of datalog programs, we say that $\mathcal{D}$ is a parallelly tractable datalog program (PTD) class if there exists an NC algorithm that performs 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 are not NC algorithms, since their computational complexity is PTime-complete. 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.

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 $E \subseteq T_R^\omega(\boldsymbol{I}) \times T_R^\omega(\boldsymbol{I})$;*

$\mathcal{G}$ satisfies the following condition:

- *$\forall H, B_1, ..., B_n \in V$ such that $e(B_1, H), ..., e(B_n, H) \in E$ and $B_1, ..., B_n$ are all the parents of H, we have that $B_1, ..., B_n \rightarrow H \in P^*$.*

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 $\mathbf{I}$. We call such a node an *explicit node*. We call the other nodes in $\mathcal{G}$ the *implicit nodes*. We say that a node v is a *single-way derivable* (SWD) node if v has at most one implicit parent node; nodes with more than one implicit parent nodes are called *multi-way derivable* (MWD) nodes. The size of $\mathcal{G}$, denoted by $|\mathcal{G}|$, is the number of nodes in $\mathcal{G}$. The depth of $\mathcal{G}$, denoted by $\mathtt{depth}(\mathcal{G})$, is the maximal length of a path in $\mathcal{G}$. We next give an example of a materialization graph.

Example 1. *Consider a DHL(∘) ontology $\mathcal{O}_{ex_1}$ where the TBox is $\{\exists R.A \sqsubseteq A\}$, the RBox is $\{S \circ R \sqsubseteq R\}$ and the ABox is $\{A(b), R(a_1, b), S(a_i, a_{i-1})\}$ for $2 \leq i \leq k$ and k 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 explicit nodes whose in-degree is 0 are the original facts in $\boldsymbol{I}$. Each of the implicit 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.*

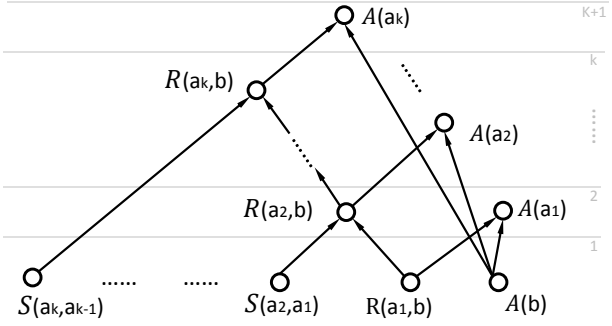

Figure 1. An example of a materialization graph.

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.

A Parallel Algorithm. In this part, we propose a parallel algorithm (Algorithm 1) that constructs a materialization graph for a given datalog program.

Algorithm 1. Given a datalog program $P = \langle R, \mathbf{I}\rangle$, the algorithm returns a materialization graph $\mathcal{G}$ of P. Recall that P^* denotes the ground instantiation of P, which consists of all possible ground instantiations of rules in R. Suppose we have $|P^*|$ processors, and each rule instantiation in P^* is assigned to one processor.[6] 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 $\mathcal{G}$,[7] 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 1 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*

[6] This might not be practically feasible, but we focus on a theoretical analysis here. In practice, one can map several rule instantiations to a single processor.

[7] 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.

$R(a_2, b)$ *are in* $\mathcal{G}_{ex_1}$*. Algorithm 1 halts when* $A(a_k)$ *has been added to* $\mathcal{G}_{ex_1}$ *(Step 3).*

Lemma 1 shows the correctness of Algorithm 1 and that, for any datalog program P, Algorithm 1 always constructs a materialization graph with the minimum depth among all the materialization graphs of P. The proofs of Lemma 1 and other lemmas and theorems can be found in the technical report.

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

1. *Algorithm 1 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 1, its depth equals the depth of the stage. □

We now discuss how Algorithm 1 can be restricted to an NC version. (I) Since Algorithm 1 does not introduce new constants and each predicate has a constant arity, one can check that $|P^*|$ is polynomial in the size of P. This also means that the number of processors is polynomially bounded. (II) The computing time of Step 1 and Step 3 occupies constant time units because of parallelism. (III) The main computation part in Algorithm 1 is the iteration of Step 2. In each iteration of Step 2, all processors work independently from each other. Thus, in theory, Step 2 costs one time unit. The whole computing time turns out to be bounded by the number of iterations of Step 2. (IV) We use the symbol ψ to denote a poly-logarithmically bounded function. The input of ψ is the size of P and the output is an non-negative integer. Based on (I, II, III, IV), for any datalog program P, if we use $\psi(|P|)$ to bound the number of iterations of Step 2, then Algorithm 1 is an NC algorithm, denoted by A_1^ψ.

Based on A_1^ψ, we can identify a class of datalog programs $\mathcal{D}_{\mathsf{A}_1^\psi}$ where all the datalog programs can be handled by A_1^ψ. It is obvious that $\mathcal{D}_{\mathsf{A}_1^\psi}$ is a PTD class.

We further show that $\mathcal{D}_{\mathsf{A}_1^\psi}$ can be captured in terms of materialization graph properties based on the following theorem.

Theorem 1. *For any datalog program* P*,* $P \in \mathcal{D}_{\mathsf{A}_1^\psi}$ *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. □

The algorithm A_1^ψ is restricted in the sense that it cannot even work on the rather simple datalog program P_{ex_1} in Example 1. The graph $\mathcal{G}_{ex_1}$ in Figure 1 is the unique materialization graph of P_{ex_1}. One can also check that $\mathtt{depth}(\mathcal{G}_{ex_1}) = k$. This means that the depth of $\mathcal{G}_{ex_1}$ is linearly bounded by k. On the other hand, the size of P_{ex_1} is a polynomial of the integer k. Thus, for any ψ that is poly-logarithmically bounded, we can always find a k large enough such that A_1^ψ terminates without constructing a materialization graph of P_{ex_1}. However there indeed exists an NC algorithm that can handle P_{ex_1}. We discuss this in the next section.

4 Two Optimized NC Algorithms

In this section, we discuss how to optimize Algorithm 1 such that P_{ex_1} can be handled. Based on the optimized variants of Algorithm 1, we can identify other PTD classes.

An Optimization Strategy. We discuss our optimization based on a general 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 1 in Figure 2. A materialization graph* $\mathcal{G}$ *is being constructed for some datalog program* $\langle R, \boldsymbol{I} \rangle$*. The bottom nodes in the dashed box are the original facts in* $\boldsymbol{I}$*. 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 *(*$1 \leq i \leq k$*) is an SWD (single-way derivable) node. The node* v' *is an MWD (multi-way derivable) node. All of the nodes* v_i *(*$1 \leq i \leq k$*) and* v' *would be added to* $\mathcal{G}$ *afterwards.*

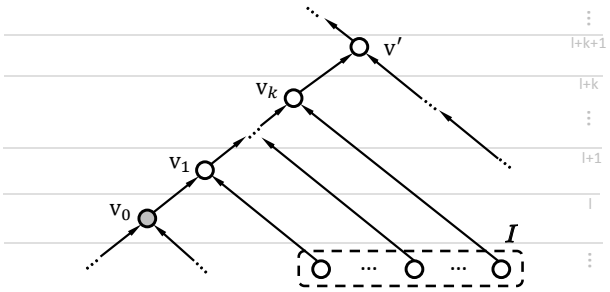

Figure 2. A partial materialization graph.

In Example 3, v_k would be added to $\mathcal{G}$ after *at least* k iterations by performing Algorithm 1. Observe that v_k is reachable from v_0 through the path $(v_0, v_1, ..., v_k)$. On the one hand, each node v_i ($1 \leq i \leq k$) can be added to $\mathcal{G}$ whenever its parent v_{i-1} is in $\mathcal{G}$, since v_i is an SWD node, i.e., v_{i-1} is the unique implicit parent node of v_i. Since v_0 has been added to $\mathcal{G}$, one can add all the nodes v_i ($1 \leq i \leq k$) to $\mathcal{G}$ right after v_0. Based on this observation, we optimize Algorithm 1 using the following strategy:

(**Strategy**) *In every iteration of Step 2, for each SWD node* v*, we add* v *to* $\mathcal{G}$ *immediately if* v *is reachable from some node that has been in* $\mathcal{G}$ *through a path containing only SWD nodes.*

For an SWD node v in some materialization graph $\mathcal{G}$, we say that a path τ is a *derivable path* of v if τ starts from some node that has been in $\mathcal{G}$ and ends in v and only contains SWD nodes. To describe the reachability between two nodes, we use a binary transitive relation $\mathtt{rch} \subseteq T_R^\omega(\mathbf{I}) \times T_R^\omega(\mathbf{I})$, e.g., $\mathtt{rch}(v_1, v_2)$ means that v_2 is reachable from v_1. In each iteration of Step 2, we compute a 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 *is not in* $\mathcal{G}$*:*

1. *if the body atoms* $B_1, ..., B_n$ *are all in* $\mathcal{G}$*, add* $rch(B_1, H), ..., rch(B_n, H)$ *to* S_{rch}*;*
2. *if* B_i *is the unique implicit node in the body and not yet in* $\mathcal{G}$*, add* $rch(B_i, H)$ *to* S_{rch}*.* □

We then compute the transitive closure of rch with respect to S_{rch}. From the transitive closure, we can identify such SWD nodes that

can be added to $\mathcal{G}$ in advance. The following algorithm applies this optimization strategy.

`Algorithm 2`. The algorithm requires two inputs: a datalog program $P = \langle R, \mathbf{I}\rangle$ and a (partial) materialization graph $\mathcal{G}$ that is 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 '$B_1, .., B_i, .., B_n \rightarrow H$' such that $\texttt{rch}(B', H)$, $\texttt{rch}(B'', B_i) \in S^*_{rch}$ where B', B'' are 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 [2]. Based on this result and Algorithm 2, we propose a variant of Algorithm 1:

`Algorithm 3`. 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 2; use an `NC` algorithm to compute the transitive closure S^*_{rch} (see (**ii**) in Algorithm 2); update $\mathcal{G}$ by performing (**iii**) in Algorithm 2.
(*Step 3*) If no node has been added to $\mathcal{G}$ (in Step 2), terminate, otherwise iterate Step 2. □

The following lemma shows the correctness of Algorithm 3.

Lemma 2. *Given a datalog program $P = \langle R, \boldsymbol{I}\rangle$, Algorithm 3 halts and outputs a materialization graph $\mathcal{G}$ of P.*

Proof sketch. This lemma is proved in two stages: (1) the graph $\mathcal{G}$ returned by Algorithm 3 is a materialization graph; (2) $\mathcal{G}$ is a complete materialization graph. We prove (1) by an induction on the iterations of Step 2 in Algorithm 3. To prove (2), we use the same method as in the proof for Lemma 1 to show that all atoms in $T^{\omega}_{R}(\mathbf{I})$ have to be added to $\mathcal{G}$. □

Example 4. *We perform Algorithm 3 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 $\{rch(R(a_{i-1}, b), R(a_i, b))|2 \leq i \leq k\} \cup \{rch(R(a_i, b), A(a_i))|1 \leq i \leq k\}$. 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 Algorithm 3 analogously to the process for Algorithm 1. It can be checked that an iteration of Step 2 in Algorithm 3 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, Algorithm 3 is an `NC` algorithm. Analogously to A^{ψ}_1, we use A^{ψ}_3 to denote an `NC` variant. Specifically, for any datalog program P, the number of iterations of Step 2 in Algorithm 3 is bounded by $\psi(|P|)$, where ψ is a poly-logarithmically bounded function.

Based on A^{ψ}_3, we can identify a `PTD` class $\mathcal{D}_{\mathsf{A}^{\psi}_3}$. The following theorem shows that $\mathcal{D}_{\mathsf{A}^{\psi}_3}$ can also be captured by the properties of a materialization graph.

Theorem 2. *For any datalog program P, $P \in \mathcal{D}_{\mathsf{A}^{\psi}_3}$ iff P has a materialization graph $\mathcal{G}$ such that the number of MWD nodes in each path of $\mathcal{G}$ is upper-bounded by $\psi(|P|)$.*

Proof sketch. ($\Rightarrow$) Suppose each materialization graph of P has a path where the number of MWD nodes is not upper-bounded by $\psi(|P|)$. This also means the number of iterations of Step 2 is not upper-bounded by $\psi(|P|)$ when constructing a materialization graph of P. Thus A^{ψ}_3 cannot handle P.

($\Leftarrow$) Suppose P has a materialization graph $\mathcal{G}$ such that the number of MWD nodes in each path is upper-bounded by $\psi(|P|)$. It is not hard to check that A^{ψ}_3 returns either of $\mathcal{G}$ or the other materialization graph $\mathcal{G}'$ that has fewer MWD nodes in each path than that of $\mathcal{G}$. □

Further Optimizing Algorithm 3. Algorithm 3 can be further optimized. In step (**i**) of Algorithm 2, when computing S_{rch}, for the rule instantiations of the form '$B_1, .., B_i, .., B_n \rightarrow H$', $B_1, ..., B_n$ (except B_i) are restricted to be explicit nodes (see (†)). We now extend S_{rch} by allowing that $B_1, ..., B_n$ (except B_i) could also be implicit nodes which have been added to the constructed materialization graph. Consider Example 3 again. If all the other implicit parents (except v_k) of v' have been added to the materialization graph, $\texttt{rch}(v_k, v')$ can also be put in S_{rch}. This allows some MWD nodes being added to the materialization graph in advance. In this way, a derivable path represents such a path where the starting node is in the constructed materialization graph and each of the other nodes (whether or not it is an SWD node) has only one parent that is not in the constructed materialization graph. Algorithm 4 is given based on this optimization.

`Algorithm 4`. This algorithm is almost the same as Algorithm 3 except Step 2. Thus we only give the new step here.

(Step 2) For all rule instantiations of the form $B_1, .., B_i, .., B_n \rightarrow H$ where H is not yet in $\mathcal{G}$, compute S_{rch} as follows:

(*1*) if all of $B_1, ..., B_n$ are in $\mathcal{G}$, add $\texttt{rch}(B_1, H)$,...,$\texttt{rch}(B_n, H)$ to S_{rch};
(*2*) if $B_1, ..., B_n$ (except B_i) are already in $\mathcal{G}$, put $\texttt{rch}(B_i, H)$ in S_{rch}.

Compute S^*_{rch}; update $\mathcal{G}$ based on S^*_{rch}. □

It is easy to prove the correctness of Algorithm 4 by referring to Lemma 2. Similarly, we use A^{ψ}_4 to denote the `NC` variant of Algorithm 4, and $\mathcal{D}_{\mathsf{A}^{\psi}_4}$ is the corresponding `PTD` class. Further, we have the following corollary. This corollary also implies that Algorithm 4 performs better than Algorithm 1 and Algorithm 3 in terms of computing time.

Corollary 1. *For any poly-logarithmically bounded function ψ, we have that $\mathcal{D}_{\mathsf{A}^{\psi}_1} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_3} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_4}$.*

Proof sketch. Suppose $P \in \mathcal{D}_{\mathsf{A}^{\psi}_1}$. According to Theorem 1, the depth of the materialization graph $\mathcal{G}$ constructed by A^{ψ}_1 is upper-bounded by $\psi(|P|)$. It is obvious that the number of MWD nodes in each path of $\mathcal{G}$ is also upper-bounded by $\psi(|P|)$. Similarly we can prove that $\mathcal{D}_{\mathsf{A}^{\psi}_3} \subseteq \mathcal{D}_{\mathsf{A}^{\psi}_4}$. □

5 Parallelly Tractable Materialization of DHL

In this section, we study whether A_4^{ψ} can handle DHL ontologies. Unfortunately there exist DHL ontologies such that A_4^{ψ} does not work. In the following, we first give such an ontology to illustrate the reason why A_4^{ψ} 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.

Path Twisting. We find that, an unlimited usage of axioms of the form $B_1 \sqcap B_2 \sqsubseteq A$ may make it impossible for Algorithm 4 to construct a materialization graph in a poly-logarithmical number of iterations of Step 2. We use the following example to illustrate it.

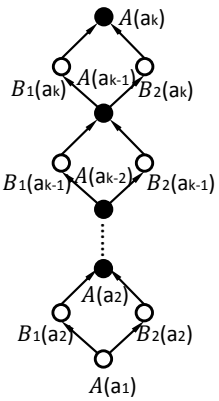

Figure 3. A partial graph of $\mathcal{G}_{ex_2}$.

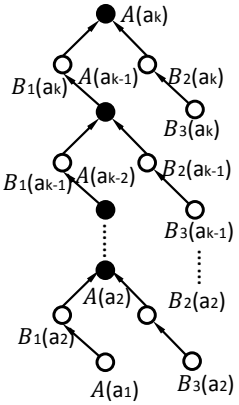

Figure 4. A partial materialization graph $\mathcal{G}_{ex_3}$.

Example 5. *Given a DHL ontology $\mathcal{O}_{ex_2}$ 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 $\mathcal{O}_{ex_2}$ by $P_{ex_2} = \langle R, \boldsymbol{I}\rangle$, 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_2} constructed by Algorithm 4 is denoted by $\mathcal{G}_{ex_2}$. Figure 3 shows a partial graph of $\mathcal{G}_{ex_2}$. Note that all binary predicates in P_{ex_2} (S and R) are EDB predicates. We include only unary atoms in Figure 3 for clarity. Further, all MWD nodes are filled with black color.*

One can check that $\mathcal{G}_{ex_2}$ is the unique materialization graph of P_{ex_2}. We focus on the partial materialization graph in Figure 3. Observe that there exists a path (e.g., $A(a_1), B_1(a_2), A(a_2), ..., A(a_k)$) involving $k-1$ MWD nodes. Obviously there is not a poly-logarithmical function ψ such that A_3^{ψ} handles P_{ex_2}. Further, when performing Algorithm 4, it can be checked that all the $k-1$ MWD nodes $(A(a_2), ..., A(a_{k-1}))$ have to be added to $\mathcal{G}_{ex_2}$ in at least $k-1$ iterations. Thus Algorithm 4 cannot handle P_{ex_2} in a poly-logarithmical number of iterations either. 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 MWD nodes. It makes the optimization of acceleration used in Algorithm 3 and Algorithm 4 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 $\mathcal{G}_{ex_2}$, there would not exist an available derivable path for $A(a_i)$. We use '*path twisting*' to represent such cases.

Note that, applying the rules corresponding to either of (T2) or (R3) can also generate MWD nodes. However, we find that these rules do not lead to the situations of 'path twisting'. We show this in the proof of Theorem 3, which can be found in our technical report.

Simple Concept. In order to make Algorithm 4 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 MWD nodes generated from the rules corresponding to (T1)*. We explain it using the following example where the ontology is modified from that in Example 5.

Example 6. *Consider an ontology 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_3} 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_3} has a unique materialization graph denoted by $\mathcal{G}_{ex_3}$. Figure 4 shows a partial graph of $\mathcal{G}_{ex_3}$ where only unary atoms are involved, and all MWD nodes are filled with black color.*

In the above example, for the axiom $B_1 \sqcap B_2 \sqsubseteq A$, all derived atoms of the form $B_2(x)$ are SWD nodes. This ensures that only one path exists between each two MWD nodes among $A(a_2), ..., A(a_k)$. Further, when constructing $\mathcal{G}_{ex_3}$, Algorithm 4 can terminate after two iterations of Step 2. Specifically, Algorithm 4 adds all SWD nodes ($B_3(a_i)$ and $B_2(a_i)$, $2 \leq i \leq k$) to $\mathcal{G}_{ex_3}$ in the first iteration; after that, all the other nodes (including MWD nodes) are added to $\mathcal{G}_{ex_3}$ in the second iteration (because each MWD node has a derivable 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 $B_2(x)$ are SWD nodes. 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 MWD ancestors of $B_1(x)$ for some x is generated from the rules corresponding to (T1).

Example 7. *In the ontology of Example 5, all of A, B_1 and B_2 are non-simple concepts. In the ontology of Example 6, 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 6 follows the simple-concept restriction and can be handled by A_4^{ψ} 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 A_4^{ψ} 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. $\mathcal{D}_{dhl} \subseteq \mathcal{D}_{\mathsf{A}_4^\psi}$.*

Proof sketch. Suppose $\mathcal{G}$ is a materialization graph of a datalog program P in $\mathcal{D}_{dhl}$. In $\mathcal{G}$, the nodes of the form $R(x, y)$ can only be derived by applying the rules corresponding to (R1-R3). All ground atoms derived from (R1) and (R2) correspond to the SWD nodes in $\mathcal{G}$. Thus, MWD nodes of the form $R(x, y)$ are only derived by applying (R3), which is to compute transitive closures. It can be checked that all binary atoms would be added to $\mathcal{G}$ in poly-logarithmically many iterations of Step 2 by performing Algorithm 4. The unary atoms of the form $A(x)$ can also be added to $\mathcal{G}$ in poly-logarithmically many iterations of Step 2 due to the simple-concept restriction. □

6 Parallelly Tractable Materialization of DHL(∘)

In this section, we study parallelly tractable materialization of DHL(∘) ontologies. In addition to the rules in DHL, we also have to consider complex RIAs (R4). In the following, we first show that complex RIAs may also cause the situation of 'path twisting'. Inspired by the simple-concept restriction, we then propose to restrict the usage of complex RIAs such that A_4^ψ works for some poly-logarithmical function ψ.

Restricting Usage of Complex RIAs. With complex RIAs, 'path twisting' may also happen when constructing a materialization graph by Algorithm 4. Consider the following example.

Example 8. *Given a DHL(∘) ontology $\mathcal{O}_{ex_4}$ 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_4} 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_4} constructed by Algorithm 4 is denoted by $\mathcal{G}_{ex_4}$.*

One can check that the materialization graph $\mathcal{G}_{ex_4}$ has the same shape as that of $\mathcal{G}_{ex_2}$ in Figure 3. A twisted path exists in $\mathcal{G}_{ex_4}$ involving $R(a_i, a_i)(2 \leq i \leq k)$ as MWD 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, at least one of R_1 and R_2 is a *simple role*.[8] 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'.[9] 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 MWD nodes of the predicate R_1 through the corresponding rule of α_3. To tackle this 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 R is not a transitive role, 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 9. *For the ontology $\mathcal{O}_{ex_4}$ in Example 8, all of the roles R_1, R_2 and R are non-simple roles. Thus, $\mathcal{O}_{ex_4}$ does not follow the simple-role restriction because of $R_1 \circ R_2 \sqsubseteq R$. Consider the ontology $\mathcal{O}_{ex_1}$ in Example 1 again. The role R is a non-simple role, while S is a simple role. Thus $\mathcal{O}_{ex_1}$ follows the simple-role restriction. All the implicit nodes in $\mathcal{G}_{ex_1}$ are SWD nodes. Thus, 'path twisting' cannot happen when materializing $\mathcal{O}_{ex_1}$ by Algorithm 4.*

We further give Theorem 4 to show that A_4^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. $\mathcal{D}_{dhl(\circ)} \subseteq \mathcal{D}_{\mathsf{A}_4^\psi}$.*

Proof sketch. The proof idea of this theorem is similar to that of Theorem 3. Specifically, we can separate the materialization of DHL(∘) ontologies into two parts: in the first part (Part 1), all the rules of the forms (R1-R4) are exhaustively applied; in the second part (Part 2), the rules of the forms (T1) and (T2) are then applied while the results of Part 1 serve as facts. In Part 1, since the rules of the form (R4) follow the simple-role restriction, it can be checked that all binary atoms would be added to the target materialization graph in a poly-logarithmical number of iterations of Step 2 by performing Algorithm 4. Part 2 can also be handled by A_4^ψ due to the simple-concept restriction. □

7 Practical Usability of the Theoretical Results

In this section, we analyze different kinds of 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[10] (*Social Network Intelligence BenchMark*), BSBM[11] (*Berlin SPARQL Benchmark*) and LODIB[12] (*Linked Open Data Integration Benchmark*). The ontology used in IIMB[13] (*The ISLab Instance Matching Benchmark*) follows the simple-concept restriction.

In the latest version of LUBM[14] (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

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

[9] 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.

[10] https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark

[11] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

[12] http://wifo5-03.informatik.uni-mannheim.de/bizer/lodib/

[13] http://islab.di.unimi.it/iimb/

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

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:

$$A(x) \rightarrow \exists y(R(x,y) \wedge B(y)) \tag{1}$$

Rule (1) introduces new anonymous constants. This kind of rule is not considered when using OWL RL reasoners to handle LUBM [30, 32]. On the other hand, in some cases, this kind of rule can also be eliminated when taking a rewriting approach [4]. In summary, if rules such as (1) are not considered, the materialization of a LUBM dataset can be handled by algorithm A_4^{ψ}.

YAGO. The knowledge base YAGO[15] is constructed from Wikipedia and WordNet and the latest version YAGO3 [17] 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 [27]. 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. However, in [27], 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,[16] Swoogle[17] and Oxford ontology lib.[18] All ontologies are available online.[19] 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.

For ontologies that satisfy the simple-concept and simple-role restrictions, users have a guarantee of parallel tractability. On the other hand, developers and users can also refer to $\mathcal{D}_{dhl}$ and $\mathcal{D}_{dhl(\circ)}$ when building their own ontologies.

8 Discussions and Related Work

Parallel reasoning with ontology languages has been extensively studied in the past decade.

The parallel reasoner RDFox [19] handles reasoning on datalog rewritable ontology languages. Algorithm 1 proposed in Section 3 is similar to the main algorithm for RDFox (see [19], Sections 3 and 4). A thread in RDFox handles several rule instantiations with respect to a fact. Such a thread corresponds to a group of processors in Algorithm 1 that is assigned with the rule instantiations handled by the thread. Thus the materialization of the datalog program in Example 1 is serial on RDFox. We use Algorithm 3 and Algorithm 4 to show that the datalog program in Example 1 is also parallelly tractable, i.e., belonging to $\mathcal{D}_{\mathsf{A}_3^{\psi}}$ and $\mathcal{D}_{\mathsf{A}_4^{\psi}}$.

The authors of [3] propose a parallel approach for RDFS encoding and reasoning and SPARQL query answering on the Cray XMT supercomputer. In [8] the authors study stream reasoning over RDF data and SPARQL query answering using Yahoo S4. The authors in [7] report their work on RDFS reasoning on massively parallel GPU hardware. In [22], the RETE algorithm is used to improve RDFS reasoning. The authors of [26] propose a more efficient storage technique and optimize the join operations in RDFS reasoning. The above works study parallel reasoning in RDFS or its fragment ρdf [20].

Distributed parallel platforms, like MapReduce or Peer-to-Peer networks, are also used for RDFS reasoning. The representative systems are WebPIE [30], Marvin [21] and SAOR [9]. Data partitioning strategies are also studied [24, 32]. To study parallel tractability on distributed platforms, we have to discuss other issues, e.g., *network structures* and *communications*. This is not considered in this work. Parallel reasoning is also implemented for other OWL fragments, e.g., OWL RL [14], OWL EL [13], OWL QL [15], and even highly expressive languages [25, 16, 23, 33]. Parallelism can also improve the performance of reasoning in non-monotonic logics [28]. Unlike the above work, the aim of our work is not to devise an efficient parallel reasoning algorithm, but to identify ontologies that are tractable for parallel materialization.

9 Conclusions and Future Work

In this paper, we studied the problem of finding ontologies such that the materialization over them is parallelly tractable. To this end, we proposed several NC algorithms 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, DHL and its extension DHL($\circ$), and proposed two restrictions such that materialization is parallelly tractable. To verify the usefulness of our theoretical results, we analyzed different kinds of datasets, including well-known benchmarks, real-world ontologies and a famous dataset YAGO. Our analysis shows that YAGO and many real ontologies belong to the parallelly tractable class $\mathcal{D}_{dhl}$ or $\mathcal{D}_{dhl(\circ)}$. On the other hand, 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.

In our future work, we will study in detail how to further apply the theoretical results in practice. One idea is to study the impact of the simple-concept and simple-role restrictions by analyzing more real-world ontologies. We also want to study parallelly tractable materialization on distributed systems. This is more challenging since several factors like network structure and communication should be taken into account. Finally, we plan to investigate the problem of parallel tractability of other OWL fragments, e.g., OWL RL and OWL EL.

Acknowledgement

We would like to thank the reviewers for their comments, which helped improve this paper considerably. Guilin Qi is supported by NSFC grant 61272378 and the 863 program under Grant 2015AA015406. Birte Glimm acknowledges the support of the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems" funded by the German Research Foundation (DFG).

[15] http://www.mpi-inf.mpg.de/home/
[16] http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library
[17] http://swoogle.umbc.edu/
[18] http://www.cs.ox.ac.uk/isg/ontologies/lib/
[19] https://github.com/quanzz/ECAI2016

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, *Foundations of Databases*, Addison-Wesley, 1995.

[2] Eric Allender, 'Reachability problems: An update', in *Proc. of CiE*, pp. 25–27, (2007).

[3] Eric L. Goodman, Edward Jimenez, David Mizell, Sinan Al-Saffar, Bob Adolf, and David J. Haglin, 'High-performance computing applied to semantic databases', in *Proc. of ESWC*, pp. 31–45, (2011).

[4] Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang, 'Acyclicity notions for existential rules and their application to query answering in ontologies', *J. Artif. Intell.*, **47**, 741–808, (2013).

[5] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*, Oxford University Press, New York, 1995.

[6] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker, 'Description logic programs: combining logic programs with description logic', in *Proc. of WWW*, pp. 48–57, (2003).

[7] Norman Heino and Jeff Z. Pan, 'RDFS reasoning on massively parallel hardware', in *Proc. Of ISWC*, pp. 133–148, (2012).

[8] Jesper Hoeksema and Spyros Kotoulas, 'High-performance Distributed Stream Reasoning using S4', in *Proc. of OOR*, (2011).

[9] Aidan Hogan, Andreas Harth, and Axel Polleres, 'Scalable authoritative OWL reasoning for the web', *Int. J. Semantic Web Inf. Syst.*, **5**(2), 49–90, (2009).

[10] Ian Horrocks and Ulrike Sattler, 'Decidability of SHIQ with complex role inclusion axioms', *J. Artif. Intell.*, **160**(1-2), 79–104, (2004).

[11] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii, 'A model of computation for mapreduce', in *Proc. of SODA*, pp. 938–948, (2010).

[12] Yevgeny 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*, pp. 2040–2045, (2009).

[13] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik, 'The incredible ELK - from polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies', *J. Autom. Reasoning*, 1–61, (2014).

[14] Vladimir Kolovski, Zhe Wu, and George Eadon, 'Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system', in *Proc. of ISWC*, pp. 436–452, (2010).

[15] Domenico Lembo, Valerio Santarelli, and Domenico Fabio Savo, 'A graph-based approach for classifying OWL 2 QL ontologies', in *Proc. of DL*, pp. 747–759, (2013).

[16] Thorsten Liebig and Felix Müller, 'Parallelizing tableaux-based description logic reasoning', in *Proc. of OTM Workshops*, pp. 1135–1144, (2007).

[17] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek, 'YAGO3: A knowledge base from multilingual wikipedias', in *Proc. of CIDR*, (2015).

[18] Robert Meusel, Christian Bizer, and Heiko Paulheim, 'A web-scale study of the adoption and evolution of the schema.org vocabulary over time', in *Proc. of WIMS*, pp. 15:1–15:11, (2015).

[19] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu, 'Parallel materialisation of datalog programs in centralised, main-memory RDF systems', in *Proc. of AAAI*, pp. 129–137, (2014).

[20] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez, 'Simple and efficient minimal RDFS', *J. Web Sem.*, **7**(3), 220–234, (2009).

[21] Eyal Oren, Kotoulas Spyros, Anadiotis George, Siebes Ronny, ten Teije Annette, and van Harmelen Frank, 'Marvin: Distributed reasoning over large-scale Semantic Web data', *J. Web Sem.*, 305–316, (2009).

[22] Martin Peters, Sabine Sachweh, and Albert Zündorf, 'Large scale rule-based reasoning using a laptop', in *Proc. of ESWC*, pp. 104–118, (2015).

[23] Anne Schlicht and Heiner Stuckenschmidt, 'Distributed resolution for ALC', in *Proc. of DL*, pp. 326–341, (2008).

[24] Ramakrishna Soma and Viktor K. Prasanna, 'A data partitioning approach for parallelizing rule based inferencing for materialized OWL knowledge bases', in *Proc. of ISCA*, pp. 19–25, (2008).

[25] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm, 'Konclude: System description', *J. Web Sem.*, **27**, 78–85, (2014).

[26] Julien Subercaze, Christophe Gravier, Jules Chevalier, and Frédérique Laforest, 'Inferray: fast in-memory RDF inference', *J. PVLDB*, **9**(6), 468–479, (2016).

[27] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum, 'YAGO: A large ontology from wikipedia and wordnet', *J. Web Sem.*, **6**(3), 203–217, (2008).

[28] Ilias Tachmazidis, Grigoris Antoniou, Giorgos Flouris, Spyros Kotoulas, and Lee McCluskey, 'Large-scale Parallel Stratified Defeasible Reasoning', in *Proc. of ECAI*, pp. 738–743, (2012).

[29] Herman 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), 79–115, (2005).

[30] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal, 'Webpie: A web-scale parallel inference engine using mapreduce', *J. Web Sem.*, **10**, 59–75, (2012).

[31] Leslie G. Valiant, 'A bridging model for parallel computation', *Commun. ACM*, 103–111, (1990).

[32] Jesse Weaver and James A. Hendler, 'Parallel materialization of the finite RDFS closure for hundreds of millions of triples', in *Proc. of ISWC*, pp. 682–697, (2009).

[33] Kejia Wu and Volker Haarslev, 'A parallel reasoner for the description logic ALC', in *Proc. of DL*, pp. 675–690, (2012).