Exploring Parallel Tractability of Ontology Materialization

Abstract

Materialization is an important reasoning service for the 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 the angle of data. That is, we aim to identify what kind of ontologies make the task of materialization parallelly tractable. In this work, we focus on datalog rewritable ontology languages. We identify two classes of datalog rewritable ontologies such that materialization over them is parallelly tractable, i.e., in `NC` complexity. We show that to determine the members in these classes is undecidable. We further give two decidable classes for the specific ontology languages RDFS and a datalog rewritable OWL fragment DHL (Description Horn Logic). We analyze two well-known datasets and show that they belong to the previous classes.

1 Introduction

The Web Ontology Language OWL[1] is an important standard for ontology languages in the Semantic Web. In the applications built on OWL, materialization is a basic service by computing all the implicit facts (or knowledge) for a given OWL ontology. Due to the generation of data by sensor networks, social media and organizations, there is an exponential growth of semantic data [Meusel *et al.*, 2015]. Thus it is challenging to perform materialization on such large-scale ontologies efficiently.

To make materialization sufficiently efficient and scalable in practice, many works present parallel reasoning systems. For example, RDFox [Motik *et al.*, 2014] is a parallel implementation for materialization of datalog rewritable ontology languages. WebPIE [Urbani *et al.*, 2012] and Marvin [Oren *et al.*, 2009] are two distributed systems for RDFS (and its extensions) reasoning. There are also works that use parallel techniques to handle scalable reasoning for highly expressive ontology languages [Schlicht and Stuckenschmidt, 2008; Wu and Haarslev, 2012]. However, according to [Raymond Greenlaw, 1995], even for RDFS and datalog rewritable ontology languages, which lead to PTime-complete or higher complexity[2] 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 [Sundara *et al.*, 2010], 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. Thus we are motivated to study the problem of making materialization efficient from the angle of data. That is, we aim to identify what kind of ontologies make the task of materialization parallelly tractable.

According to [Motik *et al.*, 2014], many real large-scale ontologies are essentially expressed in the ontology languages that can be rewritten into datalog rules. In this paper, we focus on the datalog rewritable ontology languages. Our aim is to identify the classes of datalog rewritable ontologies such that materialization over these classes is parallelly tractable, i.e., in the parallel complexity `NC`, which is studied as a complexity class where each problem can be efficiently solved in parallel [Raymond Greenlaw, 1995]. To this end, we first give an `NC` algorithm that performs materialization. We then identify a class of ontologies that can be handled by this algorithm (Section 3). We further optimize this algorithm and identify another class of ontologies based on the algorithm variant (Section 4). We show that to determine the members in those two classes is undecidable (Section 5). We further provide two decidable classes by studying the specific ontology languages, RDFS and DHL (*Description Horn Logic*) that is a datalog rewritable fragment of OWL [Grosof *et al.*, 2003] (Section 6). Based on our method, we analyze two well-known datasets, LUBM and YAGO (Section 7), and

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

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

show that they belong to the previous classes[3]. Related work is discussed in Section 8. We conclude this paper in Section 9.

2 Preliminaries

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

Datalog. Since we study the datalog rewritable ontology languages, we discuss the main issues in this paper using conventional datalog notions. In datalog [Abiteboul *et al.*, 1995], 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: $H \leftarrow B_1, ..., B_n$, 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 '$H \leftarrow$', i.e., a rule with an empty body and the head H being a ground atom. 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. θ is 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. 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*.

RDFS and DHL. Our methods can be applied to the datalog rewritable ontology languages. In particular, we study RDF Schema[4] (RDFS), and DHL (*Description Horn Logic*) that is a fragment of OWL [Grosof *et al.*, 2003]. An RDFS ontology is a set of *triples*. An example of triple is ⟨`b rdfs:type Father`⟩ that means `b` is a father. A DHL ontology consists of two parts: *terminological axioms* (TBox) and *instance assertions* (ABox). For example, an axiom `Father`$\sqsubseteq$`Person` means `Father` is a subclass of `Person`. An instance assertion `Father(b)` also says `b` is a father. We refer the readers to the works [Grosof *et al.*, 2003; Horrocks and Patel-Schneider, 2004] of rewriting RDFS and DHL ontologies into datalog programs. We make some conventions here. Suppose an RDFS ontology $\mathcal{O}$ can be rewritten into a datalog program P. A *triple* corresponds to a fact in P. The materialization rule set R for RDFS correspond to the datalog rules. For a DHL ontology, an axiom in the TBox can be rewritten into one or several datalog rules. An instance assertion corresponds to a datalog fact. For both of RDFS and DHL, we also use $\langle \mathcal{O}, R\rangle$ to represent the corresponding datalog program, *and assume that the rule set R is fixed*.

Ontology Materialization. Based on the above representations, the ontology materialization can be formalized by the evaluation of datalog programs. Specifically, given a datalog program $\langle \mathcal{O}, R\rangle$, let $T_R(\mathcal{O}) = \{H\theta | \forall H \leftarrow B_1, ..., B_n \in R, B_i\theta \in \mathcal{O}(1 \leq i \leq n)\}$, where θ is some substitution; further let $T_R^0(\mathcal{O}) = \mathcal{O}$ and $T_R^i(\mathcal{O}) = T_R^{i-1}(\mathcal{O}) \cup T_R(T_R^{i-1}(\mathcal{O}))$ for each $i > 0$. The smallest integer n such that $T_R^n(\mathcal{O}) = T_R^{n+1}(\mathcal{O})$ is called *stage*, and *materialization* refers to the computation of $T_R^n(\mathcal{O})$ with respect to $\mathcal{O}$ and R. $T_R^n(\mathcal{O})$ is also called the fixpoint and conventionally denoted by $T_R^\omega(\mathcal{O})$.

NC. The parallel complexity class `NC`, known as Nick′s Class [Raymond Greenlaw, 1995], is studied by theorists as a parallel complexity class where each decision problem can be efficiently solved in parallel poly-logarithmic time, i.e., by taking poly-logarithmic time on a PRAM (parallel random access machine) with polynomial number of processors. From the perspective of implementations, the `NC` problems are also highly parallel feasible for other parallel models like BSP [Valiant, 1990] and MapReduce [Karloff *et al.*, 2010]. `NC` complexity is originally defined as a class of decision problems. Since we study the problem of materialization, we do not restrict in this work that a problem should be a decision problem in `NC`.

3 Parallelly Tractable Class

Parallelly Tractable Class. Our target is to find what kind of ontologies make the task of materialization parallelly tractable. Since we assume that for any datalog program $\langle \mathcal{O}, R\rangle$ 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 [Raymond Greenlaw, 1995]. 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 the 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). 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 $\mathcal{A}$ for datalog materialization, then we can identify a `PTD` class $\mathcal{D}_\mathcal{A}$, which is the class of all datalog programs that can be handled by $\mathcal{A}$. 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 analysis of the given algorithm convenient.

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

[3]Here we actually considered a restricted version of LUBM.

[4]http://www.w3.org/TR/rdf-schema/

is the node set; E is the edge set; f is a mapping that maps V to $T_R^\omega(\mathcal{O})$; g is a mapping that maps V to P^*. f and g are constrained by the following condition: $\forall v, v_1, ..., v_n \in V$ such that $e(v_1, v), ..., e(v_n, v) \in E$ and $v_1, ..., v_n$ are all the parents of v, we have that $g(v) =$'$H \leftarrow B_1, ..., B_n$' and $H \leftarrow B_1, ..., B_n \in P^*$ iff $f(v) = H$, $f(v_i) = B_i (1 \leq i \leq n)$.

For some ground atom H, there may exist several rule instantiations where H occurs as a head atom. This also means H can be derived in different ways. The condition in the above definition restricts that, only one way to derive H is described by a materialization graph. Suppose $\mathcal{G}$ is a materialization graph, the nodes whose fan-in is 0 are the original facts in $\mathcal{O}$. 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 when v has at most one implicit parent node; for the nodes with more than two implicit parent nodes, we call them *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 $\mathcal{G}$ is denoted by $\texttt{depth}(\mathcal{G})$, which is the length of the longest path in $\mathcal{G}$. We give an example of materialization graph as follows.

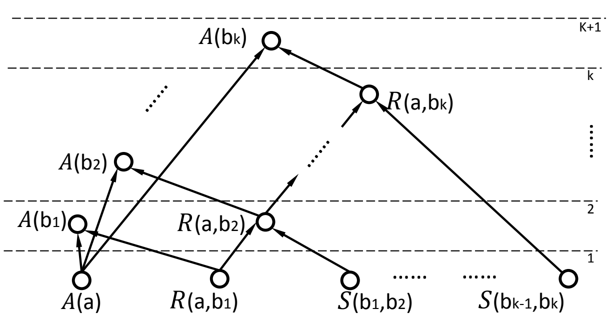

Figure 1: An example of materialization graph.

Example 1 *Consider a DHL ontology whose TBox is $\{A \sqsubseteq \forall R.A,\ R \circ S \sqsubseteq R\}$, ABox contains several assertions $\{A(a), R(a, b_1), S(b_i, b_{i+1})\}$ where $1 \leq i \leq k-1$ and k is an integer and greater than 1. The corresponding datalog program of this ontology is $P_{exp} = \langle \mathcal{O}, R \rangle$ where $\mathcal{O}$ is the ABox, R contains two rules: '$A(y) \leftarrow A(x), R(x, y)$', and '$R(x, z) \leftarrow R(x, y), S(y, z)$'. The graph in Figure 1 is a materialization graph with respect to P_{exp}, denoted by $\mathcal{G}_{exp}$. The explicit nodes whose fan-in is 0 are the original facts in $\mathcal{O}$. Each of the implicit nodes corresponds to a ground instantiation of some rule. For example, the node $A(b_k)$ corresponds to the ground rule instantiation '$A(b_k) \leftarrow A(a), R(a, b_k)$'. The size of this materialization graph is the number of the nodes, that is $3k$. The depth of $\mathcal{G}_{exp}$ is $k+1$.*

We say that a materialization graph $\mathcal{G}$ is a *complete materialization graph* when $\mathcal{G}$ contains all ground atoms in $T_R^\omega(\mathcal{O})$. 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 Naive Parallel Algorithm. In this part, we propose a naive parallel algorithm (Algorithm 1) that constructs a materialization graph for a given datalog program.

`Algorithm 1`. Given a datalog program $P = \langle \mathcal{O}, R \rangle$, this 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[5], 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*) All facts in $\mathcal{O}$ are added to $\mathcal{G}$.

(*Step 2*) For each rule instantiation $H \leftarrow B_1, ..., B_n$, if the body atoms are all in $\mathcal{G}$ while H is not in $\mathcal{G}$[6], the corresponding processor adds H to $\mathcal{G}$ and creates arcs pointing from $B_1, ..., B_n$ to H.

(*Step 3*) The algorithm iterates Step 2 until there is no processor that can add more nodes and arcs to $\mathcal{G}$. Then the algorithm terminates. □

Example 2 *We consider the datalog program P_{exp} in Example 1 again, and perform Algorithm 1 on it. Initially, all the facts ($A(a), R(a, b_1), S(b_1, b_2), ..., S(b_{k-1}, b_k)$) are added to the result $\mathcal{G}_{exp}$ (Step 1). Then in different iterations of Step 2, the remaining nodes are inserted into $\mathcal{G}_{exp}$ by different processors. For example a processor p is allocated a rule instantiation '$A(b_2) \leftarrow A(a), R(a, b_2)$'. Then, processor p adds $A(b_2)$ to $\mathcal{G}_{exp}$ after it checks that $A(a)$ and $R(a, b_2)$ are in $\mathcal{G}_{exp}$. Algorithm 1 halts when $A(b_k)$ has been added to $\mathcal{G}_{exp}$ (Step 3).*

We use Lemma 1 to show the correctness of Algorithm 1, and for any datalog program P, Algorithm 1 can always construct a materialization graph with the minimum depth among all the materialization graphs of P.

Lemma 1 *Given a datalog program $P = \langle \mathcal{O}, R \rangle$, 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.*

We next discuss how to restrict Algorithm 1 to an NC version. (I) Since we do not allow introducing new constants during materialization and each predicate has a constant arity, one can check that $|P^*|$ is polynomial in the size[7] 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

[5]Since we target to give theoretical analysis, we do not consider the practical feasibility here.

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

[7]The size of P can be seen as the number of characters used to encode P.

Step 2. In each cycle 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 now introduce a function ψ that is poly-logarithmically bounded. 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 $\mathcal{A}_1^{\psi}$.

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

We further show that $\mathcal{D}_{\mathcal{A}_1^{\psi}}$ can be captured by materialization graph based the following theorem.

Theorem 1 *For any datalog program P, $P \in \mathcal{D}_{\mathcal{A}_1^{\psi}}$ iff P has a materialization graph whose depth is upper-bounded by $\psi(|P|)$.*

$\mathcal{A}_1^{\psi}$ is restricted in a sense that it cannot even work on the datalog program P_{exp} in Example 1. The graph $\mathcal{G}_{exp}$ in Figure 1 is the unique materialization graph of P_{exp}. One can also check that depth($\mathcal{G}_{exp}$)= $k+1$. This means that the depth of $\mathcal{G}_{exp}$ is linearly bounded by k. On the other hand, the size of P_{exp} can be captured by k. Thus for any ψ that is poly-logarithmically bounded, we can always find a large k such that $\mathcal{A}_1^{\psi}$ terminates without constructing a materialization graph of P_{exp}. However there indeed exists an NC algorithm that can handle P_{exp}. We discuss this in the next section.

4 An Optimized NC Algorithm

In this section, we optimize Algorithm 1 such that P_{exp} can be handled. Based on the optimized variant of Algorithm 1, we can identify the other PTD class.

We find that, in the case of Example 1, the construction of a materialization graph can be accelerated. In Example 1, $A(b_k)$ would be added to $\mathcal{G}_{exp}$ after *at least* k iterations by performing Algorithm 1. Observe that $A(b_k)$ is reachable from $R(a,b_1)$ through the path $(R(a,b_1), R(a,b_2), ..., R(a,b_k))$. On the one hand, each node $R(a,b_i)$ $(2 \leq i \leq k)$ can be added to $\mathcal{G}_{exp}$ whenever its parent $R(a,b_{i-1})$ is in $\mathcal{G}$, since $R(a,b_i)$ is an SWD node, i.e., $R(a,b_{i-1})$ is the unique implicit parent node of $R(a,b_i)$ (see the paragraph after Definition 2). On the other hand, $R(a,b_1)$ is initially added to $\mathcal{G}_{exp}$. Thus one can add all the nodes $R(a,b_i)$ $(2 \leq i \leq k)$ and $A(b_k)$ to $\mathcal{G}_{exp}$ right after $R(a,b_1)$. Based on this observation, we can 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.*

To describe the reachability between two nodes, we use a binary transitive relation rch$\subseteq T_R^{\omega}(\mathcal{O}) \times T_R^{\omega}(\mathcal{O})$, e.g., rch$(v_1,v_2)$ means v_2 is reachable from v_1. In each cycle of Step 2, we compute a rch relation (denoted by S_{rch}) by performing the following process:

($\dagger$) *For each rule instantiation $H \leftarrow B_1,..,B_i,..,B_n$ where H is not in $\mathcal{G}$:*

(1) if the body atoms $B_1,...,B_n$ are all in $\mathcal{G}$, we add rch(B_1,H),...,rch(B_n,H) to S_{rch};

(2) if in the body, B_i is the unique implicit node and not yet in $\mathcal{G}$, we 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. We give the following algorithm to apply the optimization strategy:

Algorithm 2. This algorithm accepts two inputs: one is a datalog program $P = \langle \mathcal{O}, R \rangle$, the other one is a partial materialization graph $\mathcal{G}$ that is being constructed from P. Then the following steps are performed:

(**i**) the algorithm first computes a rch relation S_{rch} by following the above process (see ($\dagger$)).

(**ii**) the algorithm then computes the transitive closure of S_{rch} and gets a new one S_{rch}^*.

(**iii**) Finally, $\mathcal{G}$ is updated as follows: for any rch$(B_i,H) \in S_{rch}$ that corresponds to $H \leftarrow B_1,..,B_i,..,B_n$ such that rch(B',H),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, the algorithm adds H and B_i (if B_i is not in $\mathcal{G}$) to $\mathcal{G}$, and creates edges $e(B_1,H),...,e(B_n,H)$ in $\mathcal{G}$. For other statements rch$(B_j,H) \in S_{rch}$, the algorithm does nothing. □

It is well known that there is an NC algorithm for computing transitive closure [Allender, 2007]. Motivated by this result, we propose a variant of Algorithm 1 based on Algorithm 2.

Algorithm 3. Given a datalog program $P = \langle \mathcal{O}, R \rangle$, this algorithm returns a materialization graph $\mathcal{G}$ of P. Initially $\mathcal{G}$ is empty. The following steps are then performed:

(*Step 1*) All facts in $\mathcal{O}$ are added to $\mathcal{G}$.

(*Step 2*) The algorithm computes S_{rch} by performing (**i**) in Algorithm 2; the transitive closure S_{rch}^* is computed by an NC algorithm (see (**ii**) in Algorithm 2); $\mathcal{G}$ is updated by performing (**iii**) in Algorithm 2.

(*Step 3*) The algorithm iterates Step 2 until there is no node that can be added to $\mathcal{G}$. Then it terminates. □

We give the following lemma to show the correctness of Algorithm 3.

Lemma 2 *Given a datalog program $P = \langle \mathcal{O}, R \rangle$, Algorithm 3 halts and the output $\mathcal{G}$ is a materialization graph of P.*

Example 3 *We perform Algorithm 3 on the datalog program P_{exp} in Example 1. Initially, $R(a,b_1)$ is in the materialization graph $\mathcal{G}_{exp}$. In the first iteration of Step 2, all the rule instantiations are in two kinds of forms: '$A(b_i) \leftarrow A(a), R(a,b_i)$' and '$R(a,b_i) \leftarrow R(a,b_{i-1}), S(b_{i-1},b_i)$' $(2 \leq i \leq k)$, S_{rch} is the set $\{rch(R(a,b_{i-1}), R(a,b_i)) | 2 \leq i \leq k\} \cup \{rch(R(a,b_i), A(b_i)) | 1 \leq i \leq k\}$. In the transitive closure of S_{rch}, one can check that $rch(R(a,b_1), R(a,b_i))$,$rch(R(a,b_1), A(b_i)) \in S_{rch}^*(2 \leq$*

$i \leq k$). Thus $R(a, b_i)$ and $A(b_i)$ ($2 \leq i \leq k$) can all be added to $\mathcal{G}_{exp}$ in the first iteration of Step 2.

We can obtain the NC variant of Algorithm 3 as what we do for Algorithm 1. It can be checked that a cycle of Step 2 in Algorithm 3 costs poly-logarithmical 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. As the same to $\mathcal{A}_1^{\psi}$, we use $\mathcal{A}_3^{\psi}$ 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 $\mathcal{A}_3^{\psi}$, we can identify a PTD class $\mathcal{D}_{\mathcal{A}_3^{\psi}}$. One can further check that, for any ψ, $\mathcal{D}_{\mathcal{A}_1^{\psi}} \subseteq \mathcal{D}_{\mathcal{A}_3^{\psi}}$. The following theorem is given to show that $\mathcal{D}_{\mathcal{A}_3^{\psi}}$ can also be captured by materialization graph.

Theorem 2 *For any datalog program P, $P \in \mathcal{D}_{\mathcal{A}_3^{\psi}}$ 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 $\psi(|P|)$.*

5 Undecidability

We now have two PTD classes $\mathcal{D}_{\mathcal{A}_1^{\psi}}$ and $\mathcal{D}_{\mathcal{A}_3^{\psi}}$ where ψ is a poly-logarithmically bounded function. Recall that we want to find what kind of ontologies are tractable for parallel materialization. It actually requires us to check, for a given ontology, whether it belongs to some PTO class. The bad news is that, the problem of checking whether a given datalog program belongs to $\mathcal{D}_{\mathcal{A}_1^{\psi}}$ is undecidable (the same for $\mathcal{D}_{\mathcal{A}_3^{\psi}}$). We give the following theorem to show the undecidability of this problem for the previous two PTD classes.

Theorem 3 *Given any datalog program P, it is undecidable to check whether, 1) $P \in \mathcal{D}_{\mathcal{A}_1^{\psi}}$, and 2) $P \in \mathcal{D}_{\mathcal{A}_3^{\psi}}$.*

The above result indicates that although we have the two PTD classes: $\mathcal{D}_{\mathcal{A}_1^{\psi}}$, and $\mathcal{D}_{\mathcal{A}_3^{\psi}}$, we cannot identify all the datalog programs that belong to either of them.

6 Identifying Decidable Classes

In this section, we investigate two specific ontology languages: RDFS and DHL (a datalog rewritable fragment of OWL), and identify the ontologies expressed in these two languages that are tractable for parallel materialization. Instead of giving an NC algorithm at first, we propose to restrict the usage of vocabularies or terms in RDFS and DHL. In this way, we identify two classes (denoted by $\mathcal{D}_{rdfs}$ and $\mathcal{D}_{dhl}$). We prove that the algorithm $\mathcal{A}_3^{\psi}$ can handle $\mathcal{D}_{rdfs}$ and $\mathcal{D}_{dhl}$[8]. Thus $\mathcal{D}_{rdfs}$ and $\mathcal{D}_{dhl}$ are also PTD classes according to Definition 1. Furthermore, given a datalog program P, one can decide whether P belongs to either of $\mathcal{D}_{rdfs}$ and $\mathcal{D}_{dhl}$ based on the restrictions of usage. The study of $\mathcal{D}_{rdfs}$ and $\mathcal{D}_{dhl}$ is motivated by two reasons: 1) An RDFS or DHL ontology can be rewritten into a datalog program. This makes it possible to use the previous results to analyze them; 2) Several popular real datasets are essentially built on these two ontology languages, e.g., YAGO and LUBM. Thus our method can be used to analyze practical cases.

RDFS. In [Hayes, 2004], a group of rules (denoted by R_{rdfs} here) is proposed to perform materialization of RDFS ontologies. The authors of [ter Horst, 2005] proved that, in the worst case, the materialization with R_{rdfs} is in NP-complete. However, in practical applications, the usage of RDFS vocabularies should be restricted since 'illegal' statements may cause ontology hijacking [Hogan *et al.*, 2009]. Thus we follow the advices [Hogan *et al.*, 2009] of the restricted usage of RDFS vocabularies and define $\mathcal{D}_{rdfs}$ as follows:

Definition 3 *$\mathcal{D}_{rdfs}$ is a class of datalog programs in the form of $\langle \mathcal{O}, R_{rdfs} \rangle$, where the usage of vocabularies in $\mathcal{O}$ is restricted as follows:*

(1) it is not allowed to state that the domain and range of a property is `rdf:Property` *or* `rdfs:Class`*;*

(2) the statements of subclasses of `rdf:Property`, `rdfs:Class`, `rdfs:ContainerMembershipProperty` *and* `rdfs:Datatype` *are not allowed;*

(3) it is not allowed to state that `rdfs:Property` *is a subclass of some class (the same for* `rdfs:Class`, `rdfs:Literal`, `rdfs:ContainerMembershipProperty` *and* `rdfs:Datatype`*); it is not allowed to state that* `rdfs:member` *is a sub-property of some property.*

Theorem 4 *There exists a function ψ s.t. $\mathcal{D}_{rdfs} \subseteq \mathcal{D}_{\mathcal{A}_1^{\psi}}$.*

DHL. DHL [Grosof *et al.*, 2003] is essentially based on the description logic $\mathcal{SHOIQ}$ DL. For any axiom in the form of $C \sqsubseteq D$, C and D can be atomic concepts, or conjunctions ($C \sqcap D$). In particular, C can also be a disjunction ($C \sqcup D$) or an existential restriction ($\exists R.C$), while D can also be a universal restriction ($\forall R.C$). Furthermore, the statements about roles, i.e., role inclusions ($R \sqsubseteq S$), inverse roles ($R \sqsubseteq S^-$), transitivity ($R \circ R \sqsubseteq R$) and role compositions ($R \circ S \sqsubseteq T$), are also allowed in DHL. The assertions in an ABox of a DHL ontology are of two forms $C(a)$ and $R(a, b)$, which correspond to the facts in a datalog program. Readers can refer to Example 1 for the example of datalog rules rewritten from a DHL ontology. In the following, we define a class of datalog programs $\mathcal{D}_{dhl}$ and show that $\mathcal{D}_{dhl}$ is a PTD class by Theorem 5.

Definition 4 *$\mathcal{D}_{dhl}$ is a class of datalog programs that follows two conditions:*

1) each datalog program in $\mathcal{D}_{dhl}$ is rewritten from a DHL ontology;

2) for the rules rewritten from role compositions, only two kinds of rules are allowed: '$R_1(x, z) \leftarrow R_1(x, y), R_2(y, z)$' and '$R_1(x, z) \leftarrow R_2(x, y), R_1(y, z)$' where R_2 is an EDB predicate.

Theorem 5 *There exists a function ψ s.t. $\mathcal{D}_{dhl} \subseteq \mathcal{D}_{\mathcal{A}_3^{\psi}}$.*

7 Practical Usability of Theoretical Results

In this section, we analyze two well-known datasets: LUBM and YAGO. These two datasets have been widely regarded

[8]Theorem 5 is given to show that $\mathcal{D}_{rdfs}$ can even be handled by $\mathcal{A}_1^{\psi}$.

as large-scale datasets and used in many applications and projects. Based on the analysis of these two datasets, we find that, without importing other data sources, they all belong to $\mathcal{D}_{dhl}$.

LUBM. The Lehigh University Benchmark (LUBM) is a famous benchmark in the community of Semantic Web [Guo *et al.*, 2005], and has been widely used to facilitate the evaluation of ontology-based systems in a standard and systematic way. In the latest version[9], there are 48 classes and 32 properties. Most of the statements about classes can be rewritten into datalog rules that are allowed in $\mathcal{D}_{dhl}$. There are five classes that are also defined in the form of $A \sqsubseteq \exists R.B$. If we rewrite this axiom into a logic rule, it should be like:

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

The rule (1) is obtained by introducing new anonymous constants. This kind of rules are always ignored in practical reasoning when handling LUBM [Urbani *et al.*, 2012; Weaver and Hendler, 2009]. Furthermore, the statements about properties, such as inverse property statement, can be rewritten into datalog rules allowed in $\mathcal{D}_{dhl}$. In summary, if the rules like (1) are ignored, the materialization of a LUBM dataset can be handled by the algorithm $\mathcal{A}_3^{\psi}$.

YAGO. YAGO[10] is a huge knowledge base, which is constructed from Wikipedia and WordNet. The latest version YAGO3 [Mahdisoltani *et al.*, 2015] has more than 10 million entities (like 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 [Suchanek *et al.*, 2008]. In addition to the expressiveness of RDFS, YAGO *model* also allows stating the *transitivity* and *acyclicity* of a property. A group of materialization rules is also given [Suchanek *et al.*, 2008]. All the rules are allowed in $\mathcal{D}_{dhl}$. Thus we have that a well-constructed YAGO dataset belongs to $\mathcal{D}_{dhl}$.

The above two datasets model the real world knowledge and turn out to be parallelly tractable for materialization. This is also supported from the experimental perspectives [Urbani *et al.*, 2012; Sundara *et al.*, 2010]. On the other hand, the developers and users can also refer to $\mathcal{D}_{rdfs}$ and $\mathcal{D}_{dhl}$ 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 [Motik *et al.*, 2014] handles reasoning on datalog rewritable ontology languages. Algorithm 1 proposed in Section 3 is similar to the main algorithm for RDFox (see [Motik *et al.*, 2014], 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 are 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 to show that the datalog program in Example 1 is also parallelly tractable, i.e., belonging to $\mathcal{D}_{\mathcal{A}_3^{\psi}}$.

For RDFS materialization, different parallel techniques and platforms are used. The representative systems are WebPIE [Urbani *et al.*, 2012], Marvin [Oren *et al.*, 2009] and SAOR [Hogan *et al.*, 2009]. The data partition strategies are also studied [Soma and Prasanna, 2008; Weaver and Hendler, 2009]. Parallel reasoning has also been verified to be available on other OWL fragments, like OWL EL [Kazakov *et al.*, 2014], OWL QL [Lembo *et al.*, 2013], and even highly expressive languages [Liebig and Müller, 2007; Schlicht and Stuckenschmidt, 2008; Wu and Haarslev, 2012]. Parallelism can also improve the performance of evaluation on non-monotonic logic [Tachmazidis *et al.*, 2012]. Unlike the above work, our work does not aim to give parallel reasoning algorithms, but to identify such ontologies that are tractable for parallel materialization.

In other related areas, there is also work on studying how to make the target problems tractable in parallel. In the area of logic programming, the works [Ullman and Gelder, 1988; Afrati and Papadimitriou, 1993] focus on logic rules and analyze the cases where reasoning on chain-style rules is in NC. The work of [Fan and Huai, 2014] studies the problem of query answering on big data. The authors propose several classes of queries that can lead to NC algorithms on big data.

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 first proposed two 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 NC. We showed that to decide whether a given datalog program belongs to either of these PTD classes is undecidable. We further studied two specific ontology languages RDFS and DHL, and identified two decidable PTD classes. To verify the usefulness of our theoretical results, we analyzed two well-known datasets, LUBM and YAGO, which have a good performance for parallel reasoning. Our analysis shows that YAGO and a restricted version of LUBM belong to $\mathcal{D}_{dhl}$.

In one of our future works, we will study how to apply theoretical results into practice. One idea is to study the impact of the restrictions in Definition 3 and Definition 4 by analyzing more real ontologies, for example DBpedia[11] and the biomedical ontology SNOMED CT[12]. The other idea is to adapt the optimizations given in Algorithm 3 to existing parallel reasoning algorithms. Another line of future work is to extend our results to other OWL languages, e.g., OWL RL.

References

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

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

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

[11]http://wiki.dbpedia.org/

[12]http://www.ihtsdo.org/snomed-ct

1995.

[Afrati and Papadimitriou, 1993] Foto N. Afrati and Christos H. Papadimitriou. The parallel complexity of simple logic programs. *J. ACM*, 40(4):891–916, 1993.

[Allender, 2007] Eric Allender. Reachability problems: An update. In *Proc. of CiE*, pages 25–27, 2007.

[Fan and Huai, 2014] Wenfei Fan and Jinpeng Huai. Querying big data: Bridging theory and practice. *J. Comput. Sci. Technol.*, 29(5):849–869, 2014.

[Grosof *et al.*, 2003] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proc. of WWW*, pages 48–57, 2003.

[Guo *et al.*, 2005] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[Hayes, 2004] Patrick Hayes. Rdf semantics. In *W3C Recommendation*, 2004.

[Hogan *et al.*, 2009] 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.

[Horrocks and Patel-Schneider, 2004] Ian Horrocks and Peter F. Patel-Schneider. A proposal for an owl rules language. In *Proc. of WWW*, pages 723–731, 2004.

[Karloff *et al.*, 2010] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proc. of SODA*, pages 938–948, 2010.

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

[Lembo *et al.*, 2013] Domenico Lembo, Valerio Santarelli, and Domenico Fabio Savo. A graph-based approach for classifying OWL 2 QL ontologies. In *Proc. of DL*, pages 747–759, 2013.

[Liebig and Müller, 2007] Thorsten Liebig and Felix Müller. Parallelizing tableaux-based description logic reasoning. In *Proc. of OTM Workshops*, pages 1135–1144, 2007.

[Mahdisoltani *et al.*, 2015] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. YAGO3: A knowledge base from multilingual wikipedias. In *Proc. of CIDR*, 2015.

[Meusel *et al.*, 2015] 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*, pages 15:1–15:11, 2015.

[Motik *et al.*, 2014] 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*, pages 129–137, 2014.

[Oren *et al.*, 2009] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed reasoning over large-scale Semantic Web data. *J. Web Sem.*, 2009.

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

[Schlicht and Stuckenschmidt, 2008] Anne Schlicht and Heiner Stuckenschmidt. Distributed resolution for ALC. In *Proc. of DL*, pages 326–341, 2008.

[Soma and Prasanna, 2008] Ramakrishna Soma and Viktor K. Prasanna. A data partitioning approach for parallelizing rule based inferencing for materialized OWL knowledge bases. In *Proc. of ISCA*, pages 19–25, 2008.

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

[Sundara *et al.*, 2010] Seema Sundara, Medha Atre, Vladimir Kolovski, Souripriya Das, Zhe Wu, Eugene Inseok Chong, and Jagannathan Srinivasan. Visualizing large-scale RDF data using subsets, summaries, and sampling in oracle. In *Proc. of ICDE*, pages 1048–1059, 2010.

[Tachmazidis *et al.*, 2012] Ilias Tachmazidis, Grigoris Antoniou, Giorgos Flouris, Spyros Kotoulas, and Lee McCluskey. Large-scale Parallel Stratified Defeasible Reasoning. In *Proc. of ECAI*, pages 738–743, 2012.

[ter Horst, 2005] 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.

[Ullman and Gelder, 1988] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[Urbani *et al.*, 2012] 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.

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

[Weaver and Hendler, 2009] Jesse Weaver and James A. Hendler. Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In *Proc. of ISWC*, pages 682–697, 2009.

[Wu and Haarslev, 2012] Kejia Wu and Volker Haarslev. A parallel reasoner for the description logic ALC. In *Proc. of DL*, pages 675–690, 2012.