QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases

Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal

Department of Computer Science, Vrije Universiteit Amsterdam,
{j.urbani, frank.van.harmelen, schlobac, he.bal}@few.vu.nl

Abstract. Both materialization and backward-chaining as different modes of performing inference have complementary advantages and disadvantages.
Materialization enables very efficient responses at query time, but at the cost of an expensive up front closure computation, which needs to be redone every time the knowledge base changes. Backward-chaining does not need such an expensive and change-sensitive precomputation, and is therefore suitable for more frequently changing knowledge bases, but has to perform more computation at query time.
Materialization has been studied extensively in the recent semantic web literature, and is now available in industrial-strength systems. In this work, we focus instead on backward-chaining, and we present an hybrid algorithm to perform efficient backward-chaining reasoning on very large datasets expressed in the OWL Horst ($pD*$) fragment.
As a proof of concept, we have implemented a prototype called QueryPIE (Query Parallel Inference Engine), and we have tested its performance on different datasets of up to 1 billion triples. Our parallel implementation greatly reduces the reasoning complexity of a naive backward-chaining approach and returns results for single query-patterns in the order of milliseconds when running on a modest 8 machine cluster.
To the best of our knowledge, QueryPIE is the first reported backward-chaining reasoner for OWL Horst that efficiently scales to a billion triples.

1 Introduction

We are witnessing an exponential growth of semantically annotated data available on the Web. While a few years ago a large RDF dataset would consist of a few hundred thousand triples, now a large dataset is in the order of billions of triples. This growth calls for knowledge-base systems that are able to efficiently process large amounts of data.

The community has provided tools to perform efficient materialization (i.e. calculate the forward closure) using distributed techniques that can scale up to hundreds of billion statements over reasonably expressive logics [14] but there are use cases in which this technique is neither desirable nor possible. In particular, when datasets are frequently updated, materialization is not efficient.

Currently, there is no alternative to materialization that scales to relatively complex logics and very large data sizes. Backward-chaining reasoning, which does not require materialization, suffers from more complex query evaluation that adversely affects performance and scalability. Thus, it has until now been limited to either small datasets (usually in the context of expressive DL reasoners) or weak logics (RDFS inference).

To overcome this problem, we propose an *hybrid* method to perform backward-chaining reasoning that calculates some derivations in a forward fashion while the majority are computed on-the-fly during query time as necessary. This method strikes a balance between the large pre-processing costs of materialization and the complexity of pure backward-chaining reasoning. Thus, it allows us to do more complex reasoning with competitive performance. Furthermore, our algorithms have been designed to exploit the computational power of a compute cluster.

The costs of reasoning depend on the logic we consider. In this paper, we will consider the OWL Horst fragment [13], also known as the $pD*$ ruleset. OWL Horst is the most widely used complex fragment in Web-scale data to date, as witnessed by our datasets which combine some of the most important parts of the Linked Data Cloud.

Our method abstracts from the actual query language by describing and evaluating the reasoning system in terms of retrieving triples that match a given pattern. As a proof of concept, we have implemented a prototype called QueryPIE and tested its performance. QueryPIE has been built on top of the Ibis framework [1] and it was launched on the DAS-4 cluster with up to 8 machines. As we will describe later in the paper, the results indicate that our algorithms manage to keep the query response time in the order of a few milliseconds over triple stores of up to a billion statements.

The rest of the paper is organized as follows: In Section 2 we formalize our problem while in Section 3 we give a brief overview of rule-based reasoning, positioning our approach within this field. Next, in Section 4, we describe our algorithms for performing efficient backward-chaining reasoning introducing key optimizations. In Section 5 we evaluate the performance of the QueryPIE prototype on real and benchmark data. Finally, in Sections 6 and 7 we report on related work and we draw our conclusions.

2 Querying complex Web-scale data

In this paper, we consider a scenario where a user queries a potentially huge and rapidly changing knowledge base with information modeled in some expressive ontology language. Even for a simple SPARQL such as

```
SELECT ?s WHERE { ?s :lives 'Amsterdam' . ?s rdf:type Person . }
```

additional implicit information can be derived according to the formal semantics of the underlying representation language and those consequences are commonly retrieved from the knowledge base through some form of reasoning. In our case, we will study the specific problem when reasoning is invoked at query-time, i.e. at the moment when the system searches for information in the knowledge base for all triples that match `(?s :lives 'Amsterdam')` and `(?s rdf:type Person)`. In this paper, we only consider simple conjunctions of triples in a query, which means that we can assume that the input of the reasoning process is a single triple pattern.

We will use the following simple definitions: as usual, a *triple* is a sequence of three RDF terms, and a *triple pattern* is a sequence of three elements where each of them is either a variable (in this case it is preceded by a '?') or an RDF term. A *query* is a triple pattern. A *ground triple pattern* is a triple pattern not containing any variables (i.e. a ground triple pattern is a triple). Triple pattern P_1 is *more specific* than triple pattern P_2 (written ($P_1 < P_2$) if P_1 can be constructed from P_2 by replacing all occurrences of at least one variable with an RDF term.

Formally, the problem that we are addressing is the following: given a set of axioms in the language OWL Horst (which we will call the knowledge base KB) and a query Q as input, we want to derive all the ground triples $T < Q$ that are logically entailed by KB (see [13] for the definition of the entailment relation in OWL Horst).

The most common form of reasoning in ontology languages such as OWL Horst or OWL 2 RL is rule-based. It has been shown that all triples that are entailed by a OWL Horst knowledge base are precisely those triples derivable by the repeated application of a restricted set of rules defined by the language. In the following section we will review some of the rule-based reasoning approaches and the drawback with the current techniques with respect to our problem, which lead to the development of the novel approach described in Section 4.

3 Rule-based reasoning

For the purpose of this paper, we consider rule-based reasoning as a process that exhaustively applies a set of rules to a set of triples to infer some conclusions. Rules can be applied either in a forward or in a backward way. The first case is referred as materialization (or forward-chaining) while the second is referred as backward-chaining.

With materialization, the rules are applied over the entire KB until all possible triples are derived, irrespective of the input query. The main advantage of this method is that querying is simple and efficient after the closure has been calculated since it does not require further inference. The main disadvantage is that the closure needs to be updated at every change in the KB and this becomes problematic when the KB is updated frequently or when queries are infrequent compared to updates.

With backward-chaining, the rules are applied only over the strictly necessary data that lead to the derivation of ground triples of the input query. Since reasoning is only performed for the given query, updating the knowledge base is cheap because there is no closure that needs to be recomputed. Unfortunately, this flexibility comes at a price: the system has to perform specific computations for every query.

In this paper we focus on backward-chaining since currently there is no valid technique that can scale to a large extent. We define *backward-chaining reasoning* (or simply backward-chaining) over a ruleset R as a process that takes as input a triple pattern Q (the query) and a knowledge base KB and returns as output a set of triples C (the conclusions) such that each $C_i \in C$ can be derived from KB using the rules in R and $C_i < Q$ (conclusions are instantiations of the query). We call C the *answer-set* of the query Q: all triples $C_i < Q$ that are entailed by the KB by using R.

We consider the rules in the RDFS [6] and OWL Horst fragments. We report in Table 1 the rules in these two fragments because we will frequently refer to them. As we can see from this table, all rules have one or more triple patterns as antecedents and exactly one consequent.

Regardless of the set of considered rules, backward-chaining first searches for all rules with a consequent that is either compatible (i.e. contains variables in the same position) or more specific than the query pattern. After this, it will recursively look at the antecedents of these rules, regarding them as new query patterns. In this way the reasoning process builds an *and-or tree* of all the possible rules that might return some derivations.

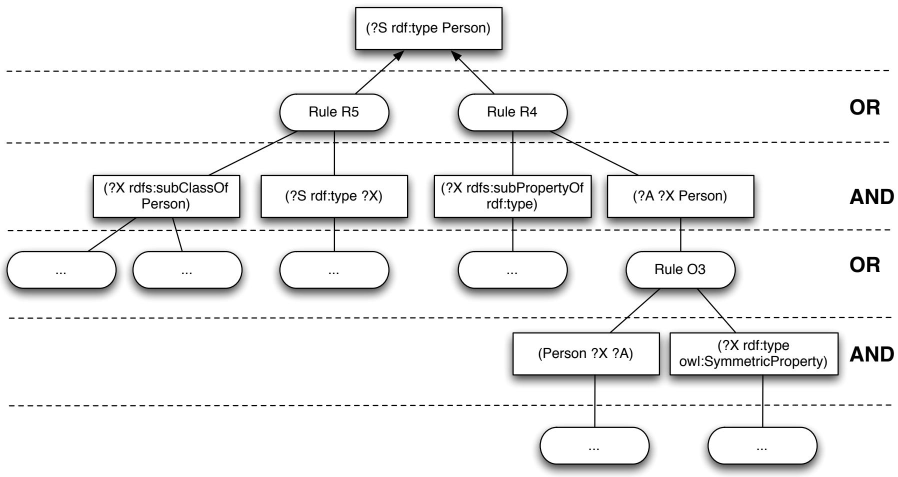

Fig. 1. Example of and-or reasoning tree

In Figure 1 we show an example of an and-or tree. Here, the derived triples are generated by different rules (the OR level) only if all of their antecedents are bound while respecting the shared variables (the AND level). The variable bindings will be propagated to the higher levels until they reach the top of the tree and can be returned as part of the answer-set. The reasoner dynamically

	Antecedents		**Consequent**
R1:	p rdfs:domain x, s p o	$\Rightarrow$	s rdf:type x
R2:	p rdfs:range x, s p o	$\Rightarrow$	o rdf:type x
R3:	p rdfs:subPropertyOf q, q rdfs:subPropertyOf r	$\Rightarrow$	p rdfs:subPropertyOf r
R4:	s p o, p rdfs:subPropertyOf q	$\Rightarrow$	s q o
R5:	s rdf:type x, x rdfs:subClassOf y	$\Rightarrow$	s rdf:type y
R6:	x rdfs:subClassOf y, y rdfs:subClassof z	$\Rightarrow$	x rdfs:subClassOf z
O1:	p rdf:type owl:FunctionalProperty, u p v , u p w	$\Rightarrow$	v owl:sameAs w
O2:	p rdf:type owl:InverseFunctionalProperty, v p u, w p u	$\Rightarrow$	v owl:sameAs w
O3:	p rdf:type owl:SymmetricProperty, v p u	$\Rightarrow$	u p v
O4:	p rdf:type owl:TransitiveProperty, u p w, w p v	$\Rightarrow$	u p v
O5:	v owl:sameAs w	$\Rightarrow$	w owl:sameAs v
O6:	v owl:sameAs w, w owl:sameAs u	$\Rightarrow$	v owl:sameAs u
O7a:	p owl:inverseOf q, v p w	$\Rightarrow$	w q v
O7b:	p owl:inverseOf q, v q w	$\Rightarrow$	w p v
O8:	v rdf:type owl:Class, v owl:sameAs w	$\Rightarrow$	v rdfs:subClassOf w
O9:	p rdf:type owl:Property, p owl:sameAs q	$\Rightarrow$	p rdfs:subPropertyOf q
O10:	u p v, u owl:sameAs x, v owl:sameAs y	$\Rightarrow$	x p y
O11a:	v owl:equivalentClass w	$\Rightarrow$	v rdfs:subClassOf w
O11b:	v owl:equivalentClass w	$\Rightarrow$	w rdfs:subClassOf v
O11c:	v rdfs:subClassOf w, w rdfs:subClassOf v	$\Rightarrow$	v rdfs:equivalentClass w
O12a:	v owl:equivalentProperty w	$\Rightarrow$	v rdfs:subPropertyOf w
O12b:	v owl:equivalentProperty w	$\Rightarrow$	w rdfs:subPropertyOf v
O12c:	v rdfs:subPropertyOf w, w rdfs:subPropertyOf v	$\Rightarrow$	v rdfs:equivalentProperty w
O13a:	v owl:hasValue w, v owl:onProperty p, u p w	$\Rightarrow$	u rdf:type v
O13b:	v owl:hasValue w, v owl:onProperty p, u rdf:type v	$\Rightarrow$	u p w
O14:	v owl:someValuesFrom w, v owl:onProperty p, u p x, x rdf:type w	$\Rightarrow$	u rdf:type v
O15:	v owl:allValuesFrom u, v owl:onProperty p, w rdf:type v, w p x	$\Rightarrow$	x rdf:type u

Table 1. *RDFS* and *OWL Horst* rulesets

builds such a tree until no rule can be further applied, or when the triple patterns can be read from the knowledge base.

Since such reasoning is executed at query time, it must be efficient, and it is crucial that this unfolding of the and-or tree is limited as much as possible. Therefore, we are required to come up with some optimizations to reduce the size of the and-or tree. In the next section, we will present some algorithms to reduce the size of the tree and hence the execution time.

4 Optimizations for backward-chaining reasoning

In this section, we propose two main optimizations that aim to reduce the and-or tree complexity. These optimizations are:

– Precompute some reasoning branches that will appear often in the tree to avoid their recomputation at query-time.
– Encourage early failure of branches, allowing to prune the and-or tree, by using the precomputed branches.

4.1 Precomputation of reasoning branches

If we look at the and-or tree in Figure 1, we notice that the execution of some reasoning branches depends less on the input than others. For example, the pattern `(?X rdfs:subPropertyOf rdf:type)` is more generic than `(?A ?X :Person)`, because the latter refers to a specific term from the query. Another difference between these two patterns is that the first corresponds to only terminological triples while the second can match any triple. It was already empirically verified [15] that terminological triples are far less than the others on Web-data, therefore, the first pattern will match with many fewer triples than the other.

We make a distinction between these two types of patterns, calling the first terminological triple patterns. A *terminological triple pattern* is a triple pattern which has as predicate or object a term from either the RDFS or the OWL vocabularies.

These terminological patterns are responsible for a notable computational cost that affects many queries. If we precompute all the ground instances of these triple patterns that are entailed by the knowledge-base, then whenever the reasoner needs such patterns it can use the precomputed results avoiding to perform additional computation. This would simplify our task to only having to perform reasoning on the non-terminological patterns. We call this simplified form of reasoning *terminology-independent reasoning* since it can avoid reasoning over terminological patterns.

Algorithm 1 Terminology-independent reasoning algorithm

```
ti-reasoner(Pattern pattern):
 //Get rules where pattern is more specific than rule's consequent
 Rules applicableRules = ruleSet.applicable(pattern)

 Results results = {}
 for(Rule rule in applicableRules)
   Patterns antecedents = rule.instantiate_antecedents(pattern)
   for(Pattern antecedent : antecedents) //Perform reasoning to fetch all antecedents
     if (antecedent != terminological)
       antecedents.add(ti-reasoner(antecedent)) //Recursive call to the reasoner
     antecedents.add(KnowledgeBase.read(antecedent))
   results += rule.apply_rule(antecedents) //Apply the rule using the antecedents triples

 return results
```

The terminology-independent reasoning algorithm is reported in pseudocode in Algorithm 1. The terminology-independent reasoner will be faster than the standard one, because it can avoid the reasoning on terminological patterns but this algorithm is only complete if all entailed instances of these terminological triple patterns have been added to the knowledge-base.

So now the problem becomes how to calculate all implied terminological triples so that the terminology-independent reasoning is complete. Such pre-computation cannot be calculated using traditional forward-chaining techniques because the complexity of the ruleset is such that completeness cannot be reached unless we calculate the entire closure. For this task backward-chaining is more appropriate but we have explained before that a naive approach does not scale for its excessive computation requisites. To solve this issue and improve the performance, we propose an algorithm to calculate the implied terminological triples using the terminology-independent reasoner in an iterative manner. This algorithm is reported in pseudocode in Algorithm 2. The first step in this method consists of listing all the terminological patterns that should be calculated beforehand. Such a list depends on the ruleset and in our case these patterns are reported in Table 2.

Then, the algorithm starts querying the knowledge base with the terminology-independent reasoner using each pattern in the table. If the reasoner will produce some derivation it will be immediately added to the knowledge base. This process is repeated until no new triples can be inferred.

By querying the reasoner using the terminological patterns, we perform the reasoning necessary to calculate the implicit terminological triples. Since the derivation of a terminological triple might require other ground triples of other terminological patterns that might not have been found yet, we need to repeat this operation adding new derivations to the knowledge base until saturation.

In this way, the reasoning tree that leads to the derivation of an implicit terminological triple is built bottom-up. The first time the system will derive only the terminological triples that require only the existence of explicit ground triples while other triples that depend on other implicit terminological triples will be missed. However, at the next iteration the system will be able to use the implicit triples derived before to infer new conclusions and reach completeness when all queries return an empty set of results.

Algorithm 2 Closure of the terminological triple patterns

```
terminological_closure():
  do {
    InferredTriples = {}
    for (Pattern pattern in terminological-patterns)
        InferredTriples += ti-reasoner(pattern)
    KnowledgeBase = KnowledgeBase + InferredTriples
  } while (InferredTriples is not empty)
```

At that point, the answer-set for all terminological queries has been computed. We will call this the *terminological closure*. After the terminological closure is completed, the terminology-independent reasoner will be sound and com-

`(?X rdfs:subPropertyOf ?Y)`	`(?X rdfs:subClassOf ?Y)`
`(?X rdfs:domain ?Y)`	`(?X rdfs:range ?Y)`
`(?P rdf:type owl:FunctionalProperty)`	`(?X owl:sameAs ?Y)`
`(?P rdf:type owl:InverseFunctionalProperty)`	`(?X owl:inverseOf ?Y)`
`(?P rdf:type owl:TransitiveProperty)`	`(?X rdf:type owl:Class)`
`(?P rdf:type owl:SymmetricProperty)`	`(?X rdf:type owl:Property)`
`(?X owl:equivalentClass ?Y)`	`(?X owl:onProperty ?Y)`
`(?X owl:hasValue ?Y)`	`(?X owl:equivalentProperty ?Y)`
`(?X owl:someValuesFrom ?Y)`	`(?X owl:allValuesFrom ?Y)`

Table 2. Terminological triple patterns considered for RDFS and OWL Horst fragment

plete. A proof of these two properties goes beyond the scope of this paper but is available online[1].

4.2 Prune reasoning using the precomputed branches

The pre-calculation of the terminological closure allows us to implement another optimization that can further reduce the size of the and-or tree by identifying beforehand whether a rule can contribute to derive facts for the parent branch.

In this case, the triples in the terminological closure can be used for the purposes of inducing early failures: the truth of these triples is easy to verify since they have been precomputed and no inference is needed. Therefore, when scheduling the derivation of rule-antecedents, we give priority to antecedents that potentially match these precomputed triples so that if these cheap antecedents do not hold, the rule will not apply anyway, and we can avoid the computation of the more expensive antecedents of the rule for which further reasoning would have been required.

To better illustrate this optimization, we proceed with an example. Suppose we have the and-or tree described in Figure 1. In this tree, the reasoner fires rule O3 (concerning symmetric properties in OWL) to be applied on the second antecedent of rule R4.

In this case, Rule O3 will fire only if some of the subjects of the triples part of `(?X rdfs:subPropertyOf rdf:type)` will also be the subject of triples part of `(?X rdf:type owl:SymmetricProperty)`. Since both patterns are more specific than terminological patterns, we know beforehand all the possible '?X', and therefore we can immediately perform an intersection between the two sets to see whether this is actually the case. If there is an intersection, then the reasoner proceeds executing rule O3, otherwise it can skip its execution since it will never fire.

It is very unlikely that the same property appears in all the terminological patterns, therefore by performing such intersections we are able to further reduce the tree size not considering rules that will derive no conclusion.

[1] `http://www.few.vu.nl/~jui200/papers/tr-iswc2011.pdf`

5 Evaluation

To evaluate the methods described above, we have implemented a proof-of-concept prototype called *QueryPIE* using the Java language and the Ibis framework [1].

The Ibis framework provides a set of libraries which ease the development of a parallel distributed application. We have used it to develop a distributed reasoner which can work on a variable number of nodes. The data is indexed with 4 indexes (*spo*, *sop*, *pos*, and *ops*) and partitioned across the nodes. The nodes load the data in the main memory, and, when the reasoner is invoked with an input pattern, it builds the and-or tree and executes it on the relevant data in the compute nodes. Then, the data is collected to one location and returned to the user. We have used the Hadoop MapReduce framework [4] and WebPIE [14] to create the data indexes and compute the sameAs closure and consolidation, which is a common practice among reasoners [14]. All the code is publicly available[2].

A complete evaluation of our work is beyond the scope of this article. In this section we report the results of a number of experiments that aimed to understand the effectiveness and performance of our algorithms. The evaluation of other aspects like scalability is left as future work.

The evaluation was performed on the DAS4 cluster[3]. Each node in this cluster is equipped with at least 24 GB of main memory, two quad-core processors and 2 TB disk space. The nodes use a 10 Gbit/s ethernet connection. The input consists of triples encoded in N-Triples format. Initially, we have compressed the datasets using the technique presented in [16]. All tests were performed using 8 machines.

We have used three datasets for our experiments, one benchmark tool and two real-world datasets. The artificial benchmark tool that we used is LUBM [5]. LUBM allows the generation of arbitrary numbers of triples. In our experiments we have generated a dataset of about 1 billion triples. The other two real-world datasets that we used are LLD[4] and LDSR[5]. The first dataset is a collection of biomedical data, taken from different sources, and it contains about 700 million triples. LDSR is a collection of generic information, of about 860 million triples. LDSR and LLD are among the largest single collections of triples that are currently available on the Web of Data. All three datasets make use of OWL modeling primitives.

Unfortunately, there is no standard set of queries on real-world datasets to benchmark the reasoner[6]. Therefore, we had to choose a number of input patterns and execute them in our prototype. In Table 3 we report the list of input patterns used in this evaluation and refer to them through this section.

[2] `http://few.vu.nl/~jui200/files/querypie-1.0.0.tar.gz`

[3] `http://www.cs.vu.nl/das4`

[4] LinkedLifeData, available at `http://linkedlifedata.com/`

[5] Also known as FactForge, available at `http://factforge.net/`

[6] Standard sets of queries exist for artificial datasets such as LUBM which we use, but not for real-world datasets

Pattern	Dataset	Pattern
1	LUBM	? ? University0
2	LUBM	University0 hasAlumnus ?
3	LUBM	? rdf:type ResearchGroup
4	LUBM	UndergradStudent0 rdf:type ?
5	LDSR	? rdf:type opencyc:Business
6	LDSR	dbpedia:Arnold_Sch...gger ? ?
7	LDSR	? rdf:type umbel:CompactCar
8	LDSR	dbpedia:Lamborghini owl:sameAs ?
9	LLD	? rdf:type gene:Gene
10	LLD	? uniprot:pathway399...145 ? ?
11	LLD	? ? skos:definition
12	LLD	? rdf:type biopax:sequenceFeature

Table 3. List of the input patterns used in the evaluation

These example query patterns were chosen because:

– they all require inference (i.e. none of them appears in the datasets) with the deliberate exception of pattern nr. 9;
– they are identical or similar[7] to the query-patterns that would be generated from the SPARQL queries that come as examples with LLD and LDSR;
– they differ in the size of the answer-set that they generate;
– they differ in the size of the inference tree that is required to derive them.

Initially we focused on the effectiveness of our algorithms and we calculated the reduction of the and-or tree size caused by our optimizations on a set of queries. The results and a more complete discussion are reported in subsection 5.1.

After this, we performed some experiments to evaluate whether the performance of our method is competitive when compared to materialization, which is currently the de-facto reasoning method over large data. Such comparison was chosen because to the best of our knowledge there is no other OWL Horst backward-chaining reasoner that works on large amounts of RDF data, and therefore we are unable to perform a comparison of the absolute reasoning performance of our algorithms. The results are discussed in subsection 5.2.

5.1 Effectiveness: comparison against naive backward-chaining

The main scope of our work was to reduce the size of the and-or tree as described in sections 4.1 and 4.2, in order to decrease the runtime of the reasoning.

The optimizations and algorithms that we described are crucial to perform backward-chaining reasoning on large data. Therefore, in order to evaluate their impact on the performance, we manually calculated how large the tree would be if we did not make any pre-calculation. For this purpose, every time the reasoner

[7] Some queries were slightly changed to evaluate different types of reasoning

Input pattern	#leaves with/out optimiz.	Ratio
Pattern 1	21/174	8.29
Pattern 2	5/58	11.60
Pattern 3	2/3	1.50
Pattern 4	38/291	7.66

Table 4. Estimated performance gain against naive backward-chaining

had to process a pattern that was already pre-calculated, we added the tree that was needed to calculate it during the initial phase. This method is in fact an underestimate, because we could not deactivate the other optimization, therefore in reality the gain is even higher than the one calculated.

Table 4 reports the actual number of leaves of the and-or tree with and without the pre-calculation. The last column reports the obtained reduction ratio and shows that the number of leaves (= the number of paths in the tree) shrinks by one order of magnitude through our pre-calculation. This shows that our pre-calculation is indeed very effective. For a very small cost in both data space and upfront computation time (see table 5), we substantially reduce the search tree. Apparently, the pre-calculation precisely captures small amounts of inferences that contribute substantially to the reasoning costs because they are being used very often.

As expected, we notice that the reduction ratio is not constant but changes depending on the input pattern. Overall, the optimized reasoning algorithm generated an and-or tree that is between 11.6-1.5 times smaller.

5.2 Performance: comparison against full materialization

With materialization, typically the data provider computes the entire closure of the data beforehand and then loads the input and derived data into a database-like infrastructure where the users can query the data with no reasoning performed on the fly. Here, we can distinguish two phases: the first, where the entire closure is computed, and the second, where the user can query the data.

In this scenario, the first phase takes a lot of time because all the reasoning must be performed while the second is much faster since only a lookup is performed. Instead, in our case the first phase will be much faster, since we do not calculate the entire closure but only a very small part that can be used to speed up the reasoning later, but the second phase will be slower since we do perform some reasoning.

In Table 5 we report the reasoning execution time of our pre-calculated closure against the execution time of calculating the entire closure for the datasets that we consider. We have used WebPIE to compute the closure on the same number of machines since it supports the same ruleset and has the best performance on large data [14]. In this table, in the first part we report respectively the runtime and the number of triples derived in the preprocessing stage of our algorithm while in the second we report the same when we compute the entire

Input	Terminological closure		Full material.	
	Time (sec.)	# statms.	Time (sec.)	# statms.
LDSR (862M)	89	0.62M	10036	927M
LLD (694M)	332	7.06M	3931	330M
LUBM (1101M)	8	22	4526	495M

Table 5. Comparison computation terminological closure against full materialization

Pattern	#Results	#leaves and-or tree	Time query back. reas. (ms.)	Time query full closure (ms.)	Ratio
Pattern 1	75613	312	55.12	35.75	1.54
Pattern 2	37118	5	38.91	17.47	2.24
Pattern 3	2400836	2	1166.84	1017.85	1.15
Pattern 4	4	38	3.53	1.02	3.46
Pattern 5	26440	411	34.83	13.57	2.57
Pattern 6	4937	60	8.57	2.86	2.99
Pattern 7	182	3	3.38	3.32	1.02
Pattern 8	5	23	3.49	0.92	3.79
Pattern 9	4524379	1	1685.55	1680.87	1.00
Pattern 10	4	134	8.17	1.10	7.43
Pattern 11	0	72	7.00	1.01	6.93
Pattern 12	245831	4	100.89	98.97	1.02

Table 6. Performance comparison at query-time of our method against full-closure approach

closure. From the table we observe, as expected, that our method is considerably faster than a traditional forward reasoner performing reasoning in about 5 minutes in the worst case against the almost 3 hours necessary to compute the closure.

Thus, even in the worst case (LLD), the costs of our preprocessing stage are only a fraction of computing the full closure, using the fastest approach for closure computation known in the literature, and using the same hardware setup.

In the second phase (when performing the actual query) our approach will be slower than engines that query a fully computed closure, since we must do reasoning at query time and it is important to evaluate such cost.

A direct comparison of the performance with existing approaches is not appropriate since the majority of the RDF stores has a single-machine architecture and/or consider a different ruleset than ours. Therefore, since our purpose is to evaluate the overhead caused by reasoning while keeping other factors constant, we proceeded as follows. First, we launched a set of queries on our prototype using the datasets with the terminological closure calculated beforehand. After this, we loaded the full closure of the dataset (as derived with WebPIE), we completely disabled reasoning in our engine, and launched the same set of queries. In this way, we kept the infrastructure constant and indirectly made a comparison with a forward reasoning scenario using the pre-calculated derivation on the same infrastructure.

The results of this comparison are presented in Table 6. For every pattern in the input we report the number of returned results, the number of leaves of the and-or tree generated when reasoning was enabled (when querying the full closure, the size of the tree is 1), the execution time when reasoning was enabled and the execution time when reasoning was disabled with the full-closured data used as input. The last column reports the ratio between the two execution times, and represents the reasoning overhead.

We observe that the overhead varies from 1.00 to 7.43. This means that in the best case reasoning does not introduce any overhead while in the worst it slows down the response time almost 7.5 times. However, even in that worst case, the response time of the system is never more than a few milliseconds. The only exceptions are patterns nr. 3 and 9, where the transport of the large number of output triples completely dominates the calculation in both cases. Similarly as before, there is no clear correlation between the input pattern and the response time since it depends on the complexity of the reasoning involved. For example, the pattern 1 generates an and-or tree with 312 leaves and it is only 1.54 times slower whereas pattern 4 generates a tree with only 38 leaves but is 3.46 times slower.

Overall, table 6 shows that the response time of our approach is competitive with querying the forward closure, while table 5 shows that our upfront cost is anywhere from 1 to 2 orders of magnitude smaller than the upfront cost of forward reasoning.

It is interesting to evaluate when our approach becomes more attractive than a full-closure approach. To this purpose, we calculated the average response time of the selected queries for the datasets LDSR and LLD. These are respectively 12.57 and 450.40 milliseconds if reasoning is activated and 5.17 and 445.49 milliseconds when querying the full closure. We used these values to estimate how many queries a system is able to answer in a certain amount of time.

The full-closure approaches start answering the queries later because they need to wait until the closure is computed (an upfront waiting time of close to 1 hour on LLD and more than 3 hours on LDSR), while our approach can start almost immediately to answer queries with an upfront waiting time of a few minutes (see table 5). However, since the response time of our approach is slower (because of reasoning), there will be a point after which the forward approach has a lower total runtime over a large number of queries. For LDSR, it takes about 1.42 million queries in order to gain back the costs of the initial closure computation amounting to about 5 hours of continuous query-load. This means that as soon as the update frequency of the data is lower than once every 5 hours, our method will be more efficient in total runtime, and more convenient because of a much smaller upfront delay.[8]

In the case of LLD, the query times are more or less equal between both approaches, while our method does keep the advantage of having only a 5 minute startup time, instead of 1 hour. This makes our approach much more competitive

[8] This calculation assumes a maximal query-load. As soon as the query load is lower than 100% utilization, the balance shifts even more in favor of backward reasoning

than before since now the full-closure approach will become convenient only after 733 thousand queries or 91 hours without any update.

6 Related work

In previous work [15, 14], we have shown scalable RDFS and OWL materialization for datasets up to 100 billion triples. There, the MapReduce programming framework was used to encode the logic for the rulesets at hand, and a set of optimizations was introduced to improve load balancing and the efficiency of the computation. In this paper, we depart from the full forward closure and take a significant step in the direction of scalable backward-chaining reasoning.

In [17], straightforward parallel RDFS reasoning on a cluster is presented. This approach replicates all schema triples to all processing nodes, partitions instance triples arbitrarily and calculates the closure of each partition. Triples extending the RDFS schema are ignored, thus the reasoning is incomplete.

In [8], a method for distributed reasoning with EL++ using MapReduce is presented, which is applicable to the EL fragment of OWL 2. No experimental results are provided.

The work on Signal/Collect [12], introduces a new programming paradigm, targeted at handling graph data in a distributed manner. Although very promising, it is not comparable to our approach, since current experiments deal with much smaller graphs and are performed on a single machine.

The operation of passing the query bindings to the lower branches of the reasoning tree is likewise applied in the Magic Sets query rewriting technique [2] and it is commonly referred as one type of sideways information passing strategy (SIPS) [11]. However, while in the latter it is used to efficiently rewrite a query, in our case we use it to prune the reasoning branches so that it becomes effective only when combined with the schema closure.

In the context of RDF stores, in [10], backward-chaining reasoning for RDFS on 4Store is presented. The authors show how they perform RDFS reasoning on their architecture but do not report on more complex inferencing than RDFS.

The Jena RDF store [3] uses a hybrid reasoner at its core with a focus on lower expressivity logics. The data store administrator can define so-called hybrid rules which include conditions for firing rules in a backwards fashion. There are no results for using Jena with a more complex ruleset.

The Virtuoso RDF store performs incomplete RDFS and OWL rule-based reasoning. Some results are reported online[9], but no experiments are reported for scaling on the number of nodes or on datasets more complex than LUBM.

7 Conclusion and Future Work

Until now, all inference engines that can handle reasonably expressive logics over very large triple stores have deployed full materialization. In the current paper,

[9] `http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticleLUBMBenchmark`

we have broken with this mold, showing that it is indeed possible to do efficient backward-chaining over large and reasonably expressive knowledge bases. The key to our approach is two optimizations which substantially reduce the size of the search space that must be navigated by a backward-chaining inference engine.

The first optimization precomputes a small number of inferences which appear very frequently in the derivation trees. By precomputing these inferences upfront instead of during query-time, we reduce the size of the trees by an order of magnitude. This of course re-introduces some amount of preprocessing (making our work strictly speaking a *hybrid* approach), but this computation is measured in terms of minutes, instead of the hours needed for the full closure computation.

The second optimization exploits these precomputed triples, to further reduce computation. It does that by giving priority to the evaluation of antecedents that potentially match these precomputed triples. If there is no match, we can avoid calculating the other more expensive antecedents that would have required additional reasoning.

Performance analysis of our approach on three datasets varying from 0.7 billion to 1.1 billion triples shows that the query response-time for our approach is competitive with that of full materialization, with response times in the low number of milliseconds on our test query patterns, running on only a small cluster of 8 machines. The small loss of response time is offset by the great gain in not having to perform a very expensive computation of many hours before being able to answer the first query.

Obvious next steps in future work would be to investigate how our algorithms scale with the number of machines, and to understand the properties of the knowledge base that influence both the cost of the limited forward computation and the size of the inference tree. Since the proposed approach is not specifically tailored around the OWL Horst ruleset, it would be interesting to extend our prototype to support the OWL 2 RL [7] ruleset and evaluate its performance.

Also, it is worth to explore whether other techniques like memoization, other SIP strategies [11], or ad-hoc query-rewriting techniques [9] can be exploited to further improve the performance.

To the best of our knowledge, this is the first time that logically complete backward-chaining reasoning over realistic OWL Horst knowledge bases of a billion triples has been realized. Our results show that this approach is feasible, opening the door to reasoning over much more dynamically changing datasets than was possible until now.

Acknowledgments: We would like to thank Barry Bishop for reviewing our work. This work was partly supported by the LarKC project (EU FP7-215535) and by the COMMIT project.

References

1. H. E. Bal, J. Maassen, R. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, T. Kielmann, F. J. Seinstra, and C. J. H. Jacobs. Real-World Distributed Computing with Ibis. *IEEE Computer*, 2010.
2. F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of SIGACT-SIGMOD*, 1985.
3. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, 2003.
4. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, 2004.
5. Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2005.
6. P. Hayes, editor. *RDF Semantics.* W3C Recommendation, Feb. 2004.
7. B. Motik, B. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL2 Web Ontology Language: Profiles. *W3C Working Draft*, 2008.
8. R. Mutharaju, F. Maier, and P. Hitzler. A MapReduce Algorithm for EL+. In *Proceedings of the 23rd International Workshop on Description Logics (DL2010)*, 2010.
9. H. Pérez-Urbina, I. Horrocks, and B. Motik. Efficient Query Answering for OWL2. *Proceedings of the ISWC*, 2009.
10. M. Salvadores, G. Correndo, S. Harris, N. Gibbins, and N. Shadbolt. The Design and Implementation of Minimal RDFS Backward Reasoning in 4store. In *Proceedings of the ESWC*, 2011.
11. P. Seshadri, J. Hellerstein, H. Pirahesh, T. Leung, R. Ramakrishnan, D. Srivastava, P. Stuckey, and S. Sudarshan. Cost-Based Optimization for Magic: Algebra and Implementation. In *ACM SIGMOD Record*, volume 25, pages 435–446. 1996.
12. P. Stutz, A. Bernstein, and W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *Proceedings of the ISWC*, 2010.
13. H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2–3):79–115, 2005.
14. J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *Proceedings of the ESWC*, 2010.
15. J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable Distributed Reasoning using MapReduce. In *Proceedings of the ISWC*, 2009.
16. J. Urbani, J. Maassen, and H. Bal. Massive Semantic Web data compression with MapReduce. In *Proceedings of the HPDC*, 2010.
17. J. Weaver and J. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *Proceedings of the ISWC*, 2009.