RDF-SQ: Mixing Parallel and Sequential Computation for Top-Down OWL RL Inference

Jacopo Urbani[1,2(✉)] and Ceriel Jacobs[1]

[1] Department of Computer Science, VU University Amsterdam, Amsterdam, The Netherlands
{jacopo,ceriel}@cs.vu.nl
[2] Max Planck Institute for Informatics, Saarbruecken, Germany

Abstract. The size and growth rate of the Semantic Web call for querying and reasoning methods that can be applied over very large amounts of data. In this paper, we discuss how we can enrich the results of queries by performing rule-based reasoning in a top-down fashion over large RDF knowledge bases.

This paper focuses on the technical challenges involved in the top-down evaluation of the reasoning rules. First, we discuss the application of well-known algorithms in the QSQ family, and analyze their advantages and drawbacks. Then, we present a new algorithm, called RDF-SQ, which re-uses different features of the QSQ algorithms and introduces some novelties that target the execution of the OWL-RL rules.

We implemented our algorithm inside the QueryPIE prototype and tested its performance against QSQ-R, which is the most popular QSQ algorithm, and a parallel variant of it, which is the current state-of-the-art in terms of scalability. We used a large LUBM dataset with ten billion triples, and our tests show that RDF-SQ is significantly faster and more efficient than the competitors in almost all cases.

1 Introduction

The ability to derive implicit and potentially unknown information from graph-like RDF datasets [8] is a key feature of the Semantic Web. This process, informally referred to as reasoning, can be performed in several ways and for different purposes. In this paper, we focus on the application of reasoning to enrich the results of SPARQL queries [14] by deriving implicit triples that are relevant for the query, and restrict our focus to rule-based reasoning in the OWL 2 RL fragment.

In this context, one method to perform reasoning is traditionally referred to as *backward-chaining*. The main idea behind backward-chaining is to rewrite the input query in a number of subqueries whose results can be used by the rules to calculate implicit answers. Backward-chaining is often implemented with a *top-down* evaluation of the rules, where the "top" is the input query and the "bottom" consists of the queries that cannot be rewritten.

We consider the top-down algorithms designed for the Datalog language [2] because almost all rules in OWL 2 RL can be represented with this language. In

© Springer International Publishing Switzerland 2015
M. Croitoru et al. (Eds.): GKR 2015, LNAI 9501, pp. 125–138, 2015.
DOI: 10.1007/978-3-319-28702-7_8

Datalog, the most popular algorithms in this category belong to the QSQ family [2,4], which consists of a series of algorithms that implement the well-known SLD-resolution technique [9]. These algorithms differ from each other because their computation can be either recursive or iterative, tuple or set oriented, sequential or parallel, or with or without adornments.

Regardless of the chosen algorithm, a common problem of backward-chaining is that reasoning is performed under tight time constraints (since typically the user waits until the query is computed) and the computation might become too expensive to guarantee an acceptable response time. On the Web, this problem is worsened by the fact that the size of current RDF datasets increases continuously. Therefore, it is paramount to develop scalable inference techniques to provide answers to the user in a timely manner.

To address this problem, we studied the salient characteristics of the existing QSQ algorithms, and designed a new algorithm in this family – which we call *RDF-SQ* – that is tailored to the characteristics of the OWL rules and RDF datasets. This algorithm contains several novelties: First, it exploits a pre-materialization of the terminological knowledge and uses it to divide the OWL rules in four different categories depending on the number and type of literals in their body. Each category is implemented in a different way, exploiting the pre-materialization and some heuristics that hold on current RDF datasets. Second, it introduces a new rules evaluation order that interleaves the execution of rules of the first two categories with rules of the last two. The first two categories of rules are executed in parallel, with the goal of collecting as much inference as possible, while the other two are executed sequentially. In doing so, our algorithm interleaves parallel and sequential computation in order to achieve higher efficiency and better utilization of modern hardware.

We tested the performance of our implementation against QSQ-R, the most well-known implementation, and a parallel variant of it that was recently applied over very large RDF knowledge bases. We used as a test ruleset a large fragment of the OWL-RL rules. Our experiments show that RDF-SQ outperforms both algorithms significantly, allowing in this way the execution of SPARQL queries with complex inference over very large knowledge graphs with more than ten billion triples.

2 Background

We assume a basic familiarity with the RDF data model [8]. Typically, users query RDF graphs using the SPARQL language [14], which can be seen as a SQL-like language to retrieve and process sub-portions of the RDF graphs.

SPARQL is a complex and rich language, but every SPARQL query can be represented at its core as a graph pattern, and its execution can be translated into a graph matching problem [14]. In this paper, we consider the most popular type of SPARQL queries, which are the ones that can be mapped with *basic graph patterns (BGP)*. These graph patterns are simply defined as a finite set of *triple patterns*, which are members of the set $(T \cup V) \times (I \cup V) \times (T \cup V)$, where T is a finite set of RDF terms, V of variables, and $I \subseteq T$ is the set of IRIs.

We use the Datalog language to formalize the process of inferring new information from the input. Due to space constraints we only briefly introduce key concepts in this language and refer the reader to [2] for a complete overview. A generic Datalog rule is of the form $R_1(w_1) \leftarrow R_2(w_2), R_3(w_3), ..., R(w_n)$. The left-hand side of the arrow is called the *head* of the rule, while the right-hand side constitutes the rule's *body*. We call each $R_x(w_x)$ with $x \in \{1..n\}$ a *literal*, which is composed of a *predicate* (R_x) and a tuple of terms $w_x := t_1, ..., t_m$. Predicates can be either *intensional* (*idb*) or *extensional* (*edb*), and only intensional predicates can occur in the head of a rule. Each Datalog *term* can be either a variable or a constant (in this paper, variables are always indicated with capital letters to distinguish them from the constants). We call a literal a *fact* if the tuple contains only constants. We say that a fact f *instantiates* a literal l if every constant in l is equal to the constant at the same position in f, and there is an unique mapping between each variable in l and a corresponding constant at the same position in f. Consequently, the instantiation $f \leftarrow f_1, f_2, \ldots, f_n$ of a rule is a sequence of facts where the mapping from constants to variables is unique across the entire rule.

In Datalog, instantiations of rules are typically calculated through the manipulation of *substitutions*, which map variables to either constants or other variables and are calculated using special functions (called θ in this work). Sets of substitutions can be joined together ($\bowtie$) or retrieved from a database of facts using a generic function called *lookup*. An *unifier* is a special substitution between two literals that is often used to verify whether the head of a rule can produce instantiations for a given literal. A unifier that is no more restrictive than needed is called a most general unifier (MGU). In this work, we use the usual definitions of these concepts. Their formal definition, and all other Datalog concepts not explicitly defined in this paper, can be found in [2,4,21].

Given a generic database I which contains a finite set of Datalog facts and a ruleset R, we say that a fact f is an *immediate consequence* of I and R if either $f \in I$ or there exists an instantiation $f \leftarrow f_1, f_2, \ldots, f_n$ of a rule in R where all f_i are in I. Calculating all immediate consequences with a rule r is a process that we refer to as the evaluation of rule r. We define T_R as a generic operator that calculates all immediate consequences so that $T_R(I)$ contains all immediate consequences of I and R. Let $T_R^0(I) := I$, $T_R^1 = T_R(I)$ and for each $i > 0$ let $T_R^i(I) := T_R(T_R^{i-1}(I))$. Since T_R is a monotonic operator, and no new symbol is generated, there will be an i where $T_R^i(I) = T_R^{i-1}(I)$. We call this database the *fixpoint* of I and denote it with $T_R^\omega(I)$.

The goal of our work is to answer SPARQL queries over $T_R^\omega(I)$. To this end, two main techniques are normally adopted: The first technique, called *forward-chaining*, stems from fixpoint semantics and consists of calculating the entire $T_R^\omega(I)$ and then (re)using the extended database to answer the SPARQL query. Forward-chaining is often implemented with a *bottom-up* evaluation of the rules, which consists of a repetitive evaluation of the rules over augmented versions of the database. This technique has been explored extensively in literature and there are several systems that implement this type of reasoning with different degrees of expressivity [7,12,18,19,27–29].

The second technique is called *backward-chaining* (or query rewriting), and is the focus of this work. It adopts a proof-theoretic approach and calculates only the subset of $T_R^\omega(I)$ necessary to answer the query. For the purposes of query answering, backward-chaining is efficient because it does not always calculate the entire derivation like forward-chaining. For this reason, backward-chaining is adopted in large-scale RDF query engines like Virtuoso [6], 4Store [17], QueryPIE [22], or Stardog [15].

Backward-chaining is normally implemented with a *top-down* rules evaluation (a notable exception is represented by the Magic Set technique [3], which can be seen as backward-chaining performed with a bottom-up evaluation). We illustrate the functioning of a typical top-down algorithm with a small example.

Example 1. Suppose that the Datalog query $Q := (A, typ, person)$ must be evaluated using a ruleset R on a database I that contains RDF triples encoded as a ternary relation T. We report the content of I and R below:

Database I

T(a,has_grade,3), T(d,has_grade,null), T(b,has_grade,6), T(student,sc,scholar) T(c,has_grade,7), T(greater,typ,trans) T(7,greater,6), T(scholar,sc,person) T(6,greater,3), T(has_grade,dom,student)

Ruleset R

$R_1 := T(A, sc, C)$	$\leftarrow T(A, sc, B), T(B, sc, C)$
$R_2 := T(A, typ, C)$	$\leftarrow T(B, sc, C), T(A, typ, B)$
$R_3 := T(A, typ, C)$	$\leftarrow T(P, dom, C), T(A, P, B)$
$R_4 := T(A, P, C)$	$\leftarrow T(P, typ, trans),$ $T(A, P, B), T(B, P, C)$

In general, a top-down algorithm would first identify which rules might produce some answers. In our example, these are R_2, R_3, R_4. Then, it would launch a number of subqueries necessary to execute the rules. In our case, R_2 would require the results of the query $T(B, sc, person)$, R_3 of $T(P, dom, person)$, and R_4 of $T(typ, typ, trans)$. These subqueries might either trigger other rules or return no result. In our example, the first subquery would trigger R_1 which would first read $T(scholar, sc, person)$ and consequently request the subquery $T(B, sc, scholar)$ in order to calculate the triples that instantiate $T(B, sc, person)$. In our case, R_1 would return the fact $T(student, sc, person)$ to R_2, which could use this fact to issue another subquery $T(A, typ, student)$. This last subquery would trigger rule R_3, which would return the facts that a, b, c, d are students back to R_2. At this point, R_2 would use this information to infer that a, b, c, d are of type $person$ and return these facts as answers to the original query. □

Unfortunately, a major problem of backward-chaining is that the number of subqueries might become too large to provide all answers in a timely manner. To

reduce this issue, backward-chaining algorithms often use *tabling* to reduce the number of evaluations. Tabling is a particular technique of *memoization* [16] and consists of caching the results of previously evaluated queries and reuse them if needed. Tabling can be either *transient* in case it is maintained only during the query execution, or *permanent* if the results are being reused across multiple queries.

3 RDF-SQ

In Datalog, the most popular type of top-down algorithms are the ones in the QSQ family [4]. The most popular QSQ algorithm is called *QSQ-R* and was presented in 1986 [23]. QSQ-R is a recursive sequential algorithm, which exhaustively evaluates each subquery before it returns the results to the rule that generated it. In this way, it can exploit tabling efficiently.

Parallel and distributed versions of QSQ have been proposed in [1,21]. The last contribution is particularly interesting since it weakens the admissibility test and lemma resolution in SLD/AL – that is the theoretical foundation upon which QSQ-R and QoSaQ are based [25,26] – by allowing the rewriting of equivalent queries in case they do not share any parent query except the root. This choice is clearly less efficient than QSQ-R, since the latter does not perform this redundant computation, but it has the advantage that it can run in parallel without expensive synchronization.

In this landscape, our contribution consists of a new algorithm, called RDF-SQ, which is inspired by these methods but is especially designed to execute the OWL RL rules. It introduces novelties that exploit characteristics of current RDF data, and reuses features of existing QSQ algorithms in an intelligent way. We can summarize the following as its salient features:

- Rules are divided into four categories, and the evaluation of rules in each category is implemented with different algorithms;
- Both permanent and transient tabling are used extensively: Terminological knowledge is pre-materialized beforehand as described in [21], and intermediate results are cached in main memory and reused during the process;
- The algorithm interleaves sessions where the rules are evaluated in parallel, and sessions where the rules are evaluated sequentially. This strategy seeks the best compromise between tabling and parallel computation.

In the following we describe RDF-SQ in more detail. First, we describe the categorization of the rules. Then, we give an informal description and report the pseudocode. Finally, we analyze the fundamental properties of termination, soundness, and completeness.

3.1 RDF-SQ: Rule Categories

When the database is loaded, the first operation performed by the system consists of pre-materializing all triples that instantiate a number of predefined *idb*

literals using the procedure described in [21]. The goal of this procedure is to avoid a redundant inference upon the same literals during the execution of multiple queries. Therefore, we can see it as a form of permanent tabling: In fact, the results of this pre-materialization are intended to be kept in main memory so that they can be easily retrieved. During the pre-materialization, the original rules are being rewritten by replacing the predicates of the pre-materialized literals with an *edb* predicate that cannot trigger further inference. It has been shown that this rewriting is harmless (after the pre-materialization), and inference using the rewritten rules produces the same derivation(s) as with the original ones [21]. Therefore, our algorithm uses the rewritten rules instead of the original ones.

Before we describe our algorithm, we introduce a categorization of rules into four disjoint categories depending on the number and type of literals they use. We also outline how the categories of rules are implemented in our system, since these two elements are instrumental in understanding the main intuition that motivates our method. In the following, we describe each category in more detail.

Category 1. This category contains all rules that have, as body literals, a fixed number of extensional predicates. These are mostly the pre-materialized literals. An example is R_1 of Example 1. In this case, all triples that instantiate the bodies of these rules are already available in main memory. Therefore, during the rule evaluation we calculate the rule instantiations by performing a number of nested loop joins [5] followed by a final projection to construct the instantiations of the head.

Category 2. This category contains all rules that have as body literals one or more pre-materialized literals and exactly one non-materialized literal. Two examples are rules R_2 and R_3 of Example 1.

These rules are more challenging than the previous ones because their evaluation requires one relational join between a set of tuples that is available in main memory and generic triples that might reside on disk. In our implementation, the pre-materialized triples are indexed in main memory, and a hash-based join is executed as new generic triples are being fetched either from the knowledge base or from other rules.

Categories 3 and 4. The third category contains rules with two or more fixed non-materialized literals (e.g. rule R_4 of Example 1) while the fourth category contains the rules where the number of literals depend on the actual input. These are the ones that use elements of the RDF lists.

These rules are the most challenging to evaluate since they also require joins between two or more generic sets of triples that can reside both on main memory and disk. These joins are performed by first collecting all triples that instantiate the first generic literal in main memory, and then passing all acceptable values to the following generic literal using a sideways-information passing strategy [2]. This process is repeated until all generic patterns are processed. At this point a final projection is used to construct the triples that instantiate the rule's head.

3.2 RDF-SQ: Main Intuition

The system receives in input a collection of triples that constitute the input database, a set of rules, and a SPARQL BGP query. As a first operation, the system performs the pre-materialization and rewrites the initial rules as described in [21]. Then, each triple pattern in the SPARQL BGP query is retrieved in a sequence, and the bindings of each variable are passed to the following pattern using sideways information passing. After the bindings are retrieved, they are joined using an in-memory hash join.

The RDF-SQ algorithm is invoked to retrieve all triples that instantiate each triple pattern. Therefore, we can abstract the inference as a process that takes as input a query that equals to a single literal and rewrites it in multiple subqueries evaluating the rules to produce the derivations.

During this evaluation, a key difference between the categories of rules is that rules in the third and fourth categories need to collect all the results of their subqueries before they can proceed with the rest of the evaluation. If these subqueries trigger further inference, then the rule evaluator must wait until the subqueries are finished. In case the rules are evaluated by different threads, the evaluator must introduce a synchronization barrier to ensure that all subqueries have terminated. In contrast, rules in the first category can be executed independently since their input is already available in main memory, and rules in the second category do not need to wait because they can produce the derivation immediately after they receive one triple. Therefore, the evaluation of the first two categories of rules can be parallelized rather easily, while in the third and fourth categories the synchronization barriers reduce the parallelism.

In RDF-SQ we leverage this distinction and only execute rules of the first two categories in parallel. These rules are executed first, so that we can collect as much derivation as possible before we start to apply the other two rule categories, which require more computation.

3.3 RDF-SQ: Pseudocode

We report the pseudocode of RDF-SQ in Algorithm 1. To perform the parallel evaluation of the rules in the first two categories, we use the parallel version of QSQ presented in [21], which we call *ParQSQ* from now on. In our pseudocode, this algorithm is represented by the function `ParQSQ_infer` and it corresponds to the function "infer" in Algorithm 1 of [21]. For the purpose of this paper, we can see `ParQSQ_infer` as a top-down algorithm that receives in input a query Q and a list of queries already requested (called *SubQueries* in our code) and returns a number of triples that instantiate Q using the ruleset P and $\mathfrak{I} \cup Mat$ as input (notice that these variables are marked as global). Internally, this function produces the same computation as in its original version, with the only difference that in the original code *SubQueries* is a local variable, while in our version it is a global synchronized variable, so that every time a new member is added through union (e.g. in line 23), the addition is permanent. This change is necessary to implement our intended behavior of expanding each query only once, like QSQ-R.

Algorithm 1. RDF-SQ Main Algorithm. Q is the input query, $\mathfrak{I}$ is a finite set of facts, and R is the ruleset. The function returns the set of all facts that instantiate Q. $\mathfrak{I}, P, R, Tmp, Mat$ are global variables.

```
1  function rdf-sq(Q, R, ℑ)
2  P := R_12 := {r ∈ R : r.type = 1 ∨ r.type = 2}
3  R_34 := {r ∈ R : r.type = 3 ∨ r.type = 4}
4  Mat, Tmp, New := ∅
5  do
6    SubQueries := ∅
7    Mat := Tmp ∪ New ∪ Mat
8    New := New ∪ ParQSQ_infer(Q, SubQueries)
9    Mat := Mat ∪ Tmp
10   for(∀SQ ∈ SubQueries ∪ {Q})
11     if SQ was already processed in this loop
12       if all queries in SQ are processed
13         goto line 32
14       else
15         continue
16     else
17       mark SQ as processed
18     all_subst := {θ_ε}
19     for ∀r ∈ R_34 s.t. SQ is unifiable with r.HEAD
20       θ_h := MGU(SQ, r.HEAD)
21       subst := {θ_ε}
22       for ∀p ∈ r.BODY
23         tuples := ParQSQ_infer(θ_h(p), SubQueries ∪ {Q})
24         Tmp := Tmp ∪ tuples
25         subst := subst ⋈ lookup(θ_h(p), tuples)
26       end for
27       all_subst := all_subst ∪ (subst ∘ θ_h)
28     end for
29     if SQ = Q then New := New ∪ ⋃_{θ∈all_subst} {θ(SQ)}
30     else Tmp := Tmp ∪ ⋃_{θ∈all_subst} {θ(SQ)}
31   end for
32 while New ∪ Tmp ⊆ Mat ∪ ℑ
33 return New
34 end function
```

We divide the functioning of RDF-SQ in three steps. First, the rules are divided in two different ruleset categories (lines 2,3). The first ruleset is assigned to P so that it is visible to ParQSQ. Second, the algorithm applies the rules in the first two categories (line 8), and all the derivation produced is collected in Mat. Third, the rules of third and fourth types are applied sequentially on each (sub)subquery produced so far (lines 10–31), and `ParQSQ_infer` is invoked on each subquery that might be requested by these rules. Notice that the inner invocation of `ParQSQ_infer` might increase the size of $SubQueries$. Therefore, in order not to enter in a infinite loop we mark each subquery as "processed" and exit after all queries have been processed by all rules. The overall process must be repeated until no rule has derived any new triple (line 32). Finally, the program returns all explicit and inferred triples that instantiate the query Q.

3.4 RDF-SQ: Termination, Soundness, Completeness

In this paper, we limit to discuss these properties only informally since formal proofs are lengthy and can be easily obtained with slight modifications of the proofs presented in [21] for the algorithm ParQSQ.

Termination. In general, every Datalog program is guaranteed to terminate because no new symbol is being introduced [2]. Our algorithm is not an exception: Both the *lookup* and *ParQSQ* functions were proven to terminate [21], and the two loops in the program will eventually terminate because there is always a maximum numbers of facts and queries that we can construct from the domain of a given input database.

Soundness. Soundness is a property that holds if every fact returned by `rdf-sq`(Q, R, I) is contained in $T_R^\omega(I)$ and instantiates Q. In our case, this property can be verified rather easily since the derivations can be generated either by `ParQSQ_infer` or by the retrieval and union of all substitutions in lines 23 and 29. These two operations are equivalent to the operations performed by `ParQSQ_infer` to produce the conclusions. Hence they are sound due to the proof in [21].

Completeness. Completeness requires that every fact that is in $T_R^\omega(I)$ and instantiates Q is returned by `rdf-sq`(Q, R, I). Completeness is a property that has "cursed" QSQ algorithms since their inception. In fact, the original version of QSQ-R presented in [23] was found to be incomplete and was fixed by the same author and others in following publications [13,24]. Despite these fixes, later implementations of QSQ presented in [10,22] and also the widely cited version in [2] are still incomplete. A good explanation for the source of this incompleteness is reported in [11]: Basically, the mistake is in relying on the intuitive assumption that if we re-evaluate a query until fix-point during the recursive process then we eventually retrieve all answers. Unfortunately, there are cases where answers derived in previous steps cannot be exploited by the current subquery because the query is subsumed by a previous subquery, and hence not further expanded.

One solution to fix this problem is to clear the cache of precomputed subqueries either at every recursive call or only on the main loop. In our pseudocode, this operation is performed in line 6 of Algorithm 1. This guarantees that in every iteration all intermediate derivations are used in every rule evaluation, and in every iteration all unique subqueries are fully expanded by every rule at least once. Therefore, our algorithm is complete because the main loop in lines 5–32 will not exit until no more derivation has been produced.

4 Evaluation

To evaluate our contribution, we compared the performance of RDF-SQ against ParQSQ and QSQ-R. We chose the first because it has shown the best scalability [21], and the second because it is the most popular QSQ algorithm. To this end, we implemented both RDF-SQ and QSQ-R algorithms inside the QueryPIE prototype, which contains the original implementation of ParQSQ.

QueryPIE is an on-disk RDF query engine written in Java, which is freely available[1]. It is written on top of Ajira [20] – a general-purpose parallel frame-

[1] https://github.com/jrbn/querypie.

Table 1. Response time of the LUBM queries on 10B triples. The numbers in bold represent the best results.

Q	Response time (ms. or seconds if with 's')						Results (#)
	ParQSQ		QSQ-R		RDF-SQ		
	C	W	C	W	C	W	
1	759	**11**	**637**	14	763	**11**	4
3	**1.6 s**	**15**	1.8 s	41	2.4 s	**15**	6
4	**4.4 s**	93	5.1s	330	6.6 s	**55**	34
5	**9.3 s**	251	9.8 s	658	11.0 s	**82**	719
7	3.5 s	67	4.1 s	236	**2.1 s**	**51**	4
8	**165.8 s**	3.0 s	176.0	5.3 s	171.8 s	**821**	7790
10	1.2 s	56	1.2 s	128	**1.1 s**	**44**	4
11	**9.4 s**	**27**	**9.4 s**	35	9.5 s	29	224
12	26.0 s	466	26.7 s	1.2 s	**25.4 s**	**238**	15
13	-	-	4636.8 s	549.9 s	**4062.3 s**	**50.3 s**	37118

work for data intensive applications that gives the possibility of splitting computation into concurrent tasks called *chains*. We set up Ajira so that it could launch at most 8 concurrent tasks.

Testbed. We used a machine equipped with a dual quad-core CPU of 2.4 GHz, 24 GB of main memory and an internal storage of two disks of 1 TB in RAID-0. We chose to launch our experiments using the LUBM benchmark for several reasons: (*i*) LUBM is the *de facto* standard for measuring the performance of OWL reasoners over very large RDF datasets; (*ii*) it was recently used to evaluate the performance of state-of-the-art OWL reasoners (e.g. [12,18]); (*iii*) it supports challenging inference that uses rules in all four categories, and contains a representative set of queries that encode different workloads.

To allow a fair comparison between the approaches, we activated the same subset of rules and pre-materialized queries that were used in [21]. The excluded rules are mainly redundant or used to derive a contradiction (these rules cannot be activated during SPARQL answering). The only notable exclusions are the rules that handle the *owl:sameAs* semantics. However, since LUBM does not support this inference, these exclusions does not impact the performance of our implementation.

Query Response Time. We loaded an input dataset that consists of a bit more than 10 billion RDF triples (LUBM(80000)), and launched the LUBM queries with the inference performed by the three top-down algorithms. We report in Table 1 the cold and warm runtimes, and the number of results of each query. Unfortunately not all queries succeeded because QueryPIE requires that all intermediate results must fit in main memory, and this precludes the execution of

queries 2, 6, 9, and 14. Also, query 13 failed for ParQSQ: it ran out of memory after about 4 h.

We measured the runtime from the moment that the query is launched to the time where all the data is collected and ready to be returned. The cold runtime reports the runtime after the system is started. The warm runtime consists of the average of the 29 consecutive runs of the same query. These last runs are performed where all the data is in memory and no disk access is performed.

The results presented in the table give some interesting insights. First of all, we notice that the difference between cold and warm reasoning reaches two orders of magnitude. This shows that I/O has a significant impact on the performance.

Second, we notice that RDF-SQ produced the shortest warm runtime in all but one case. To better understand the behaviour, we collected additional statistics during these executions and report them in Table 2. In this table, we report the maximum amount of bytes read from disk, the number of concurrent Ajira chains produced during the execution, and the number of queries requested to the knowledge base.

We chose to record these statistics because the amount of bytes read from disk gives an indication of the I/O cost required for answering the query, while the number of Ajira tasks and queries give a rough indication of the amount of reasoning that was triggered. For example, query 1 is highly selective and triggers no reasoning: In fact, only 21 megabytes are read from disk and the number of both chains and queries is small. The most I/O intensive is query 13, where about 200GB are read from disk.

Looking at the results reported in the two tables, we can draw some further conclusions. First of all, the cold runtime is clearly limited by the I/O speed. All three algorithms read about the same amount of data, except for query 13 where ParQSQ fails. Second, considering the number of chains and queries produced, we notice how ParQSQ is generally inefficient while QSQ-R positions itself as the second most efficient algorithm. However, even though ParQSQ is less efficient than QSQ-R, its runtimes are still competitive since the warm runtime is faster than QSQ-R in almost all cases. This is due to its ability to parallelize the computation.

Performance Breakdown. It is remarkable that RDF-SQ produced the smallest number of subqueries and Ajira tasks in all cases. This highlights the efficiency of RDF-SQ when compared with the other two methods. We further investigated the reasons behind this difference and found that it is due to several factors:

– The impact of parallelism on the warm runtime is limited, since most of the execution time is taken by rules of the third and fourth category. However, parallelism brings a substantial reduction of the cold runtime. For example, the execution of query 13 on a smaller data set (LUBM(8000)) with a single processing thread produces a cold runtime of about 323 s, while if we use two threads the runtime lowers to 124 s. There is no significant difference if we increase the number of threads since two threads are enough to saturate the bandwidth.

Table 2. Statistics for the tests of Table 1. PQ stands for ParQSQ, QR for QSQ-R, while RQ abbreviates RDF-SQ.

Q.	Max MB from disk	# Tasks			# Queries		
		PQ	QR	RQ	PQ	QR	RQ
1	21	18	26	18	6	8	6
3	64	144	266	138	69	98	66
4	216	7086	3275	708	3220	1055	289
5	542	6686	2907	557	2958	990	274
7	473	4527	2635	659	2029	859	282
8	13,846	6467	2427	871	2844	798	370
10	126	4476	1529	610	2006	510	261
11	1,594	57	70	52	20	22	17
12	4,359	4999	3476	775	2257	1132	327
13	198,858	-	3112	555	-	1055	274

– Executing rules of third and fourth category in a sequential manner brings substantial benefits because at every step we can fully exploit tabling and reuse all previous derivations. This is also confirmed by the fact that QSQ-R produces comparable results with ParQSQ despite it being a sequential algorithm while the other is parallel.

5 Conclusions

Overall, our evaluation gives a first empirical evidence of how our strategy of interleaving the execution between two stages, one parallel and one sequential, is beneficial. Because of this, our algorithm produced response times that are significantly lower than other state-of-the-art algorithms using an input with ten billion triples, which can be seen as graphs with more than 10 billion edges.

In the future, we plan to do further experiments to test the performance on larger queries and on datasets with higher expressivity. Furthermore, a promising research direction consists of developing techniques which can dynamically estimate whether parallelism can bring some benefit. Finally, we plan to extend inference to SPARQL queries that encode other types of graph patterns.

To conclude, our contribution shows that complex top-down OWL inference can be applied successfully over very large collections of data. This pushes forward current boundaries, and enables the enrichment of the results of queries over RDF data on an unprecedented scale.

Acknowledgments. This project was partially funded by the COMMIT project, and by the NWO VENI project 639.021.335.

References

1. Abiteboul, S., Abrams, Z., Haar, S., Milo, T.: Diagnosis of asynchronous discrete event systems: datalog to the rescue!. In: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 358–367. ACM (2005)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases, vol. 8. Addison-Wesley, Reading (1995)
3. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 1–15. ACM (1985)
4. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (1989)
5. DeWitt, D., Naughton, J., Burger, J.: Nested loops revisited. In: Proceedings of the Second International Conference on Parallel and Distributed Information Systems, pp. 230–242. IEEE Comput. Soc. Press (1993)
6. Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer, Heidelberg (2009)
7. Heino, N., Pan, J.Z.: RDFS reasoning on massively parallel hardware. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 133–148. Springer, Heidelberg (2012)
8. Klyne, G., Carroll, J.J.: Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation (2006)
9. Lloyd, J.W.: Foundations of Logic Programming. Springer, New York (1984)
10. Madalińska-Bugaj, E., Nguyen, L.A.: Generalizing the QSQR evaluation method for horn knowledge bases. In: Nguyen, N.T., Katarzyniak, R. (eds.) New Challenges in Applied Intelligence Technologies, pp. 145–154. Springer, Heidelberg (2008)
11. Madalińska-Bugaj, E., Nguyen, L.A.: A generalized QSQR evaluation method for horn knowledge bases. ACM Trans. Comput. Logic **13**(4), 32:1–32:28 (2012)
12. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In: Proceedings of AAAI. AAAI Press (2014)
13. Nejdl, W.: Recursive strategies for answering recursive queries - The RQA/FQI strategy. In: Proceedings of the 13th International Conference on Very Large Data Bases, pp. 43–50. Morgan Kaufmann Publishers Inc. (1987)
14. Prud'Hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C recommendation (2008)
15. Pérez-Urbina, H., Rodriguez-Díaz, E., Grove, M., Konstantinidis, G., Sirin, E.: Evaluation of query rewriting approaches for OWL 2. In: Proceedings of SSWS+HPCSW (2012)
16. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs (2009)
17. Salvadores, M., Correndo, G., Harris, S., Gibbins, N., Shadbolt, N.: The design and implementation of minimal RDFS backward reasoning in 4store. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part II. LNCS, vol. 6644, pp. 139–153. Springer, Heidelberg (2011)
18. Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: WebPIE: A web-scale parallel inference engine using MapReduce. Web Semant. Sci. Serv. Agents World Wide Web **10**, 59–75 (2012)

19. Urbani, J., Margara, A., Jacobs, C., van Harmelen, F., Bal, H.: DynamiTE: Parallel materialization of dynamic RDF data. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 657–672. Springer, Heidelberg (2013)
20. Urbani, J., Margara, A., Jacobs, C., Voulgaris, S., Bal, H.: AJIRA: A lightweight distributed middleware for MapReduce and stream processing. In: 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS), pp. 545–554, June 2014
21. Urbani, J., Piro, R., van Harmelen, F., Bal, H.: Hybrid reasoning on OWL RL. Semant. Web **5**(6), 423–447 (2014)
22. Urbani, J., van Harmelen, F., Schlobach, S., Bal, H.: QueryPIE: Backward reasoning for OWL horst over very large knowledge bases. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 730–745. Springer, Heidelberg (2011)
23. Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In: Expert Database Conference, pp. 253–267 (1986)
24. Vieille, L.: A database-complete proof procedure based on SLD-resolution. In: Logic Programming, Proceedings of the Fourth International Conference, pp. 74–103 (1987)
25. Vieille, L.: From QSQ towards QoSaQ: global optimization of recursive queries. In: Expert Database Conference, pp. 743–778 (1988)
26. Vieille, L.: Recursive query processing: the power of logic. Theoret. Comput. Sci. **69**(1), 1–53 (1989)
27. Weaver, J., Hendler, J.A.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 682–697. Springer, Heidelberg (2009)
28. Zhou, Y., Nenov, Y., Cuenca Grau, B., Horrocks, I.: Complete query answering over horn ontologies using a triple store. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 720–736. Springer, Heidelberg (2013)
29. Zhou, Y., Nenov, Y., Grau, B.C., Horrocks, I.: Pay-as-you-go OWL query answering using a triple store. In: Proceedings of AAAI, pp. 1142–1148. AAAI Press (2014)