Handling owl:sameAs in RDFox via Rewriting

Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks

University of Oxford, Department of Computer Science
Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom
firstname.lastname@cs.ox.ac.ukx

Abstract. *RDFox* is a scalable, centralised, main-memory, multi-core RDF system that supports materialisation-based reasoning in OWL 2 RL and SWRL. Reasoning with the owl:sameAs property using the OWL 2 RL/RDF rules can be very inefficient, so in this paper we extend RDFox to handle owl:sameAs via *rewriting*—replacing equal resources with a single representative. The main challenge in applying this well-known idea is to effectively parallelise processing. We also show that answering SPARQL queries over the rewritten triples is not straightforward due to the multiset semantics and builtin functions. Finally, we empirically show that, on data sets commonly used in practice, rewriting can reduce materialisation times and memory consumption by orders of magnitude.

1 Introduction

RDFox [12] is a scalable reasoning and querying system for RDF. Given a datalog program, possibly obtained from an OWL 2 RL ontology [11] extended with SWRL rules [7], RDFox computes the *materialisation* (i.e., all consequences) of the program and RDF data in a preprocessing step so that queries can then be evaluated directly over the materialised triples. Systems such as QueryPIE [16], Virtuoso,[1] AllegroGraph,[2] and Stardog[3] use variants of backward chaining [1] to answer queries in fragments of OWL 2 RL without an expensive preprocessing step. However, materialisation is often preferred when the performance of query answering is critical, and it has been implemented in systems such as WebPIE [16], Oracle's RDF store [17], and OWLIM SE [3]. RDFox follows the current trends in database systems [9] and stores its data in main memory, and it uses a 'mostly lock-free' [6] algorithm to support efficient parallel processing on multi-core systems. RDFox can efficiently materialise data sets of up to a billion triples on a mid-range server with 128 GB of RAM, with a speedup of up to 13.9 over a single-threaded version when using 16 physical cores [12]. The techniques used in RDFox are complementary to the ones for shared-nothing distributed RDF systems with nontrivial communication cost between the nodes (e.g., [16, 14]): each node in a distributed system can parallelise computation and/or store data using the techniques from RDFox.

[1] http://virtuoso.openlinksw.com/
[2] http://franz.com/agraph/allegrograph/
[3] http://stardog.com/

The owl:sameAs property states equalities between resources and is a common source of inefficiency in materialisation-bases systems. OWL 2 RL/RDF [11, Section 4.3] axiomatises the semantics of owl:sameAs using rules such as $\langle s', p, o\rangle \leftarrow \langle s, p, o\rangle \wedge \langle s, \mathsf{owl{:}sameAs}, s'\rangle$ that, for each pair of equal resources r and r', 'copy' all triples between r and r'. In Section 3 we show that such 'copying' can severely impact both the materialisation time and size.

Rewriting is a well-known technique for theorem proving with equality [2, 13]. In datalog systems, rewriting consists of choosing one representative from each set of equal resources, and replacing all remaining resources in the set with the representative. Variants of this idea have been implemented in systems such as Owlgres [15], WebPIE [16], Oracle's RDF store [8], and OWLIM SE [3], and they have been shown to be very effective on practical data sets.

In this paper we discuss how to incorporate rewriting into a centralised, main-memory, multi-core system such as RDFox. To support general datalog programs in which rules can contain arbitrary resources, we must correctly update the rules as equalities are derived. Our main challenge was to ensure consistency of concurrent processing. This is usually achieved via critical sections and locking, but such solutions often do not scale well to many threads. We have developed a *'mostly' lock-free* solution: most of the time, at least one thread makes progress regardless of the other threads [6]. Lock-free algorithms are less susceptible to adverse thread scheduling than lock-based ones. Our algorithm can be seen as an optimised, tuple-at-a-time version of the seminaïve algorithm [1], and it enjoys the *nonrepetition property*: each rule instantiation is considered at most once (but the same fact can be derived multiple times via different rule instantiations).

Furthermore, we show that, due to the multiset semantics of SPARQL and builtin functions, simply evaluating a SPARQL query on the rewritten triples and then 'expanding' the answers thus obtained (i.e., by substituting all resources with equal ones in all possible ways) may not produce correct answers. We discuss how to modify SPARQL query processing so as to guarantee correctness.

We have implemented rewriting in RDFox and have evaluated its performance on several widely used data sets. Comparing RDFox with the state of the art systems is difficult: Owlgres supports only the OWL 2 QL fragment of OWL 2; WebPIE is a distributed system based on the MapReduce framework; and Oracle's database and OWLIM SE are disk-based systems. However, in our previous work [12] we have already demonstrated that RDFox is competitive with state of the art OWL 2 RL systems (when no system uses rewriting), so in this paper we just compare versions of RDFox with and without rewriting. Our results show that rewriting can improve the performance of materialisation by a factor of up to 31.1 on a single thread, and that our approach parallelises computation very well, obtaining a speedup of up to 6.7 with eight physical cores, and up to 9.6 with 16 virtual cores.[4] We also show that rewriting can reduce the number of triples by a factor of up to 7.8.

[4] Virtual cores are obtained from physical ones via *hyperthreading*, where virtual cores duplicate architectural state (such as registers), but share execution resources.

2 Preliminaries

Datalog, RDF, and OWL 2 RL. A *term* is a *resource* (i.e., a constant) or a variable. Unless otherwise stated, s, p, o, and t are terms, and x, y, and z are variables. An *atom* is a triple of terms $\langle s, p, o\rangle$ called the *subject*, *predicate*, and *object*, respectively. A *fact* (or *triple*) is a variable-free atom. A *rule* r is an implication of the form (1), where $\mathsf{h}(r) := H$ is the *head*, $\mathsf{b}(r) := B_1 \wedge \ldots \wedge B_n$ is the *body* of r, and each variable in $\mathsf{h}(r)$ also occurs in $\mathsf{b}(r)$.

$$H \leftarrow B_1 \wedge \ldots \wedge B_n \tag{1}$$

A *program* P is a finite set rules. Given finite a set of *explicit* (i.e., extensional or EDB) facts E, the *materialisation* $P^\infty(E)$ of P on E is defined as usual [1]. We use the standard notions of a *substitution* σ and composition $\sigma\tau$ of substitutions σ and τ; and $\alpha\sigma$ is the result of applying σ to a term, formula, or program α.

The OWL 2 RL [11] fragment of OWL 2 supports datalog reasoning. Two styles of OWL 2 RL reasoning are commonly used in practice. First, one can encode the ontology in RDF triples, store it with the data in a single RDF graph, and use the fixed (i.e., independent from the ontology) datalog program [11, Section 4.3]. While conceptually simple, this approach is inefficient because the fixed program's rules contain complex joins. Second, one can convert the ontology into a datalog program that depends on the ontology [4], but whose rules are shorter and simpler. This approach is complete only if the data does not contain facts such as $\langle \mathsf{rdf{:}type}, \mathsf{rdf{:}type}, \mathsf{rdf{:}type}\rangle$—an assumption commonly met in practice. Our system supports either style of reasoning, but we use the latter one in our evaluation because of its efficiency.

Parallel Materialisation in RDFox. Our algorithm [12] computes $P^\infty(E)$ using N threads of a set of explicit facts E and a program P. Set E is first copied into the set of *all* facts T, after which each thread starts updating T using a fact-at-a-time version of the seminaïve algorithm [1]. In particular, a thread selects an unprocessed fact F from T and tries to match it to each body atom B_i of each rule of the form (1) in P. For each substitution σ with $F = B_i\sigma$, the thread evaluates the partially instantiated rule $H\sigma \leftarrow B_1\sigma \wedge \cdots \wedge B_{i-1}\sigma \wedge B_{i+1}\sigma \wedge \cdots \wedge B_k\sigma$ by matching the rule's body as a query in T, and adding $H\tau$ to T for each thus obtained substitution τ with $\sigma \subseteq \tau$. The thread repeats these steps until all facts in T have been processed. Materialisation finishes if at this point all other threads are waiting; otherwise, the thread waits for more facts to become available.

To implement this idea efficiently, we store all facts in T in a single table. As usual, we encode resources using nonzero integer resource IDs in a way that allows us to use these IDs as array indexes. Furthermore, we maintain three array-based and three hash-based indexes that allow us to efficiently identify all relevant facts in T when matching a given atom. Such a scheme has two important advantages. First, the indexes allow queries (i.e., rule bodies) to be evaluated using nested index loop joins with sideways information passing. Second, arrays and hash tables are naturally parallel data structures and so they support efficient concurrent updates as parallel threads derive fresh facts.

Non-repetition of derivations is a desirable property of materialisation algorithms. For example, consider applying rule $\langle x, T, z\rangle \leftarrow \langle x, R, y\rangle \wedge \langle y, S, z\rangle$ to facts $\langle a, R, b\rangle$ and $\langle b, S, c\rangle$. A naïve version of our algorithm would select $\langle a, R, b\rangle$ from T, evaluate the partially instantiated rule $\langle a, T, z\rangle \leftarrow \langle b, S, z\rangle$, and derive $\langle a, T, c\rangle$; but then, it would select $\langle b, S, c\rangle$ from T, evaluate the partially instantiated rule $\langle x, T, c\rangle \leftarrow \langle x, R, b\rangle$, and derive $\langle a, T, c\rangle$ again. Such an algorithm would consider the same rule instance twice, which is inefficient. To avoid this drawback, after selecting an unprocessed fact F from T, our algorithm matches body atoms $B_1\sigma \wedge \cdots \wedge B_{i-1}\sigma$ in $T^{\prec F}$, and the remaining body atoms in $T^{\prec F} \cup \{F\}$, where $T^{\prec F}$ is the set of facts in T that were processed before F. In our example, assuming that $\langle a, R, b\rangle$ is processed before $\langle b, S, c\rangle$, triple $\langle a, T, c\rangle$ is derived only once, when fact $\langle b, S, c\rangle$ is selected in T.

3 Problems with owl:sameAs

In this section we highlight the problems that the owl:sameAs property poses to materialisation-based reasoners by means of an example. The semantics of owl:sameAs can be captured explicitly using program $P_{\approx}$ consisting of rules (EQ_1)–(EQ_5), which axiomatise owl:sameAs as an equivalence relation; we call a set of resources all of which are equal to each other an owl:sameAs-*clique*.

$$\langle x_i, \mathsf{owl{:}sameAs}, x_i\rangle \leftarrow \langle x_1, x_2, x_3\rangle, \text{ for } 1 \leq i \leq 3 \qquad (EQ_1)$$

$$\langle x'_1, x_2, x_3\rangle \leftarrow \langle x_1, x_2, x_3\rangle \wedge \langle x_1, \mathsf{owl{:}sameAs}, x'_1\rangle \qquad (EQ_2)$$

$$\langle x_1, x'_2, x_3\rangle \leftarrow \langle x_1, x_2, x_3\rangle \wedge \langle x_2, \mathsf{owl{:}sameAs}, x'_2\rangle \qquad (EQ_3)$$

$$\langle x_1, x_2, x'_3\rangle \leftarrow \langle x_1, x_2, x_3\rangle \wedge \langle x_3, \mathsf{owl{:}sameAs}, x'_3\rangle \qquad (EQ_4)$$

$$\mathsf{false} \leftarrow \langle x, \mathsf{owl{:}differentFrom}, x\rangle \qquad (EQ_5)$$

OWL 2 RL/RDF [11, Section 4.3] also makes owl:sameAs symmetric and transitive, but those rules are redundant as they are instances of (EQ_2) and (EQ_4).

Rules (EQ_1)–(EQ_5) can lead to the derivation of many equivalent triples, as we demonstrate next using an example program P_{ex} containing rules (R)–(F_3).

$$\langle x, \mathsf{owl{:}sameAs}, \mathsf{:USA}\rangle \leftarrow \langle \mathsf{:Obama}, \mathsf{:presidentOf}, x\rangle \qquad (R)$$

$$\langle x, \mathsf{owl{:}sameAs}, \mathsf{:Obama}\rangle \leftarrow \langle x, \mathsf{:presidentOf}, \mathsf{:USA}\rangle \qquad (S)$$

$$\langle \mathsf{:USPresident}, \mathsf{:presidentOf}, \mathsf{:US}\rangle \qquad (F_1)$$

$$\langle \mathsf{:Obama}, \mathsf{:presidentOf}, \mathsf{:America}\rangle \qquad (F_2)$$

$$\langle \mathsf{:Obama}, \mathsf{:presidentOf}, \mathsf{:US}\rangle \qquad (F_3)$$

On $P_{ex} \cup P_{\approx}$, rule (R) derives that :USA is equal to :US and :America, and rules (EQ_1)–(EQ_4) then derive an owl:sameAs triple for each of the nine pairs involving :USA, :America, and :US. The total number of derivations, however, is much higher: we derive each triple once from rule (EQ_1), three times from rule (EQ_2), once from rule (EQ_3),[5] and three three times from rule (EQ_4); thus, we

[5] Rule (EQ_1) derives the triple $\langle \mathsf{owl{:}sameAs}, \mathsf{owl{:}sameAs}, \mathsf{owl{:}sameAs}\rangle$, so we can map variable x_2 to owl:sameAs in rule (EQ_3).

get 66 derivations in total for the nine owl:sameAs triples. Analogously, rule (S) derives that :Obama and :USPresident are equal, and rules (EQ_1)–(EQ_4) derive the two owl:sameAs triples 22 times in total. These owl:sameAs triples lead to further inferences; for example, from (F_1), rules (EQ_2) and (EQ_4) infer 2×3 triples with subject :Obama or :USPresident, and object :USA, :America, or :US. Each of these six triples is inferred three times from rule (EQ_2), once from rule (EQ_3), and three times from rule (EQ_4), so we get 36 derivations in total.

Thus, for each owl:sameAs-clique of size n, rules (EQ_1)–(EQ_4) derive n^2 owl:sameAs triples via $2n^3+n^2+n$ derivations. Moreover, each triple $\langle s, p, o\rangle$ with terms in owl:sameAs-cliques of sizes n_s, n_p, and n_o, respectively, is 'expanded' to $n_s \times n_p \times n_o$ triples, each of which is derived $n_s + n_p + n_o$ times. This duplication of facts and derivations is a major source of inefficiency.

To reduce these numbers, we can choose a representative for each owl:sameAs-clique and then *rewrite* all triples—that is, replace each resource with its representative [15, 16, 8, 3]. For example, after applying rule (R), we can choose :US as the representative of :USA and :America, and, after applying rule (S), we can choose :Obama as the representative of :USPresident; then, the materialisation of P_{ex} contains only $\langle$:Obama, :presidentOf, :US$\rangle$; moreover, as we show in Section 4, the number of derivations of owl:sameAs triples drops from over 60 to just six.

Since owl:sameAs triples can be derived continuously during materialisation, rewriting must be incorporated into the reasoning algorithm. An *outdated* rule or triple contains a resource that is not a representative of itself; otherwise, the rule or triple is *current*. To ensure completeness, outdated rules and triples must be updated to current versions. Doing so in a way that promotes parallel processing while ensuring consistency is the main technical difficulty in our work.

4 Parallel Reasoning With Equality Handling

We now present our algorithm that, given a set of explicit facts E and a datalog program P, uses rewriting to compute $P^\infty(E)$ on N threads.

4.1 Intuition

We maintain a mapping ρ of resources to representatives; for α a formula, $\rho(\alpha)$ is the result of replacing each resource r in α with $\rho(r)$. To promote concurrency, we do not lock ρ when computing $\rho(\alpha)$; instead, we just require $\rho(\alpha)$ to be at least as current as ρ just before the computation. For example, if ρ is the identity as we start computing $\rho(\langle a, b, a\rangle)$, and another thread makes a' the representative of a, then $\langle a, b, a\rangle$, $\langle a', b, a\rangle$, $\langle a, b, a'\rangle$, and $\langle a', b, a'\rangle$ are all valid results.

Our algorithm is similar to the one in Section 2, but it rewrites outdated facts and rules. Dealing with the rules is nontrivial: as we explained in [12], we use an index to efficiently identify rules matching a fact, and the index may need updating when ρ changes. Updating the index in parallel would be very complex, so we use a simpler solution: we initialise $P' := P$ and index its rules;

moreover, when all threads are waiting, we recompute and reindex $P' := \rho(P)$, and we insert the updated rules (if any) into a queue R of rules for reevaluation.

The main loop of each thread now allows for three different actions. First, a thread can extract a rule from R and apply it to the set of all facts T, thus ensuring that changes to the resources in the rules are taken into account.

Second, a thread can extract and process a fact F in T. The thread first checks whether F is outdated (i.e., whether $F \neq \rho(F)$); if so, the thread removes F from T and adds $\rho(F)$ to T. If F is not outdated but is of the form $\langle a, \mathsf{owl{:}sameAs}, b\rangle$ with $a \neq b$, the thread chooses a representative of the two resources, updates ρ, and adds the other resource to a queue C. The thread derives a contradiction if F is of the form $\langle a, \mathsf{owl{:}differentFrom}, a\rangle$. Otherwise, the thread partially instantiates the rules in P' containing a body atom that matches F, and it applies such rules to T as described in Section 2.

Third, a thread can rewrite outdated facts. To avoid iteration over all facts in T, the thread extracts a resource c from the queue C of unprocessed resources and uses indexes from [12] to identify each fact $F \in T$ containing c. The thread then removes each such F from T, and it adds $\rho(F)$ to T.

Parallel modification of T can be problematic, as the following example demonstrates: (1) thread A extracts a current fact F; (2) thread B updates ρ and deletes an outdated fact F'; (3) thread A derives F' from F and writes F' into T, thus undoing the work of thread B. This could be solved via locking, but at the expense of parallelisation. Thus, instead of physically removing facts from T, we just mark them as outdated; then, when matching the body atoms of partially instantiated rules, we simply skip all marked facts. All this can be done lock-free, and we can remove all marked facts in a postprocessing step.

4.2 Formalisation

We use *short-circuit evaluation* of expressions: in 'A and B' (resp. 'A or B'), B is evaluated only if A evaluates to true (resp. false). We store all facts in a data structure T that must provide several abstract operations: $T.\mathsf{add}(F)$ atomically adds a fact F to T if F is not already present in T (marked or not), returning true if T has been changed; and $T.\mathsf{mark}(F)$ atomically marks a fact $F \in T$ as outdated, returning true if F has been changed. Also, T must provide an iterator over its facts: $T.\mathsf{next}$ atomically selects and returns a fact or returns ε if no such facts exists; $T.\mathsf{hasNext}$ returns true if T contains such a fact; and $T.\mathsf{last}$ returns the last returned fact. These operations need not enjoy the ACID properties, but they must be *linearisable* [6]: each asynchronous sequence of calls should appear to happen in a sequential order, with the effect of each call taking place at an instant between the call's invocation and response. Access to T thus does not require synchronisation via locks. Given a fact F returned by $T.\mathsf{next}$, let $T^{\prec F}$ be the facts returned by $T.\mathsf{next}$ before F, and let $T^{\preceq F} := T^{\prec F} \cup \{F\}$.

For ρ the mapping of resources to their representatives, $\rho.\mathsf{mergeInto}(d, c)$ atomically checks whether d is a representative of itself; if so, it updates the representative of all resources that d represents to the representative of c and returns true. We discuss how to implement this operation and how to compute

Algorithm 1 materialise

Global:

N: the total number of threads	W: waiting threads counter (initially 0)
P: a datalog program	P': the current program (initially P)
E: explicit facts to materialise	T: all facts (initially E)
m: a mutex variable	L: reevaluation limit (initially undefined)
run: a Boolean flag (initially true)	ρ: resource mapping (initially identity)
R: a queue of rules (initially empty)	C: a queue of constants (initially empty)

```
while run do
    if ¬evaluateUpdatedRules() and ¬rewriteFacts() and ¬applyRules() then
        increment W atomically
        acquire m
            while R.isEmpty and C.isEmpty and ¬T.hasNext and run do
                if W = N then
                    for each r ∈ P' such that r ≠ ρ(r) and ρ(r) ∉ P' ∪ R do
                        R.enqueue(ρ(r))
                    L := T.last
                    P' := ρ(P)
                    run := R.isEmpty
                    notify all waiting threads
                else
                    release m, wait for notification, acquire m
            decrement W atomically
        release m
```

$\rho(\alpha)$ in more detail in Section 4.3. Moreover, $\rho(T)$ is the *rewriting* of T with ρ, and $T^\rho := \{\langle s, p, o\rangle \mid \langle \rho(s), \rho(p), \rho(o)\rangle \in T\}$ is the *expansion* of T with ρ.

An *annotated query* is a conjunction of atoms of the form $A_1^{\bowtie_1} \wedge \cdots \wedge A_k^{\bowtie_k}$, where $\bowtie_i \in \{\prec, \preceq\}$ for each $1 \leq i \leq k$. For F a fact and σ a substitution, operation $T.\mathsf{evaluate}(Q, F, \sigma)$ returns the set containing each minimal substitution τ such that $\sigma \subseteq \tau$ and $T^{\bowtie_i F}$ contains an unmarked fact $A_i\tau$ for each $1 \leq i \leq k$. Given a conjunction of atoms $Q = B_1 \wedge \ldots \wedge B_n$, let $Q^{\preceq} := B_1^{\preceq} \wedge \ldots \wedge B_n^{\preceq}$.

Finally, for P' a set of rules and F a fact, $P'.\mathsf{rules}(F)$ returns each tuple of the form $\langle r, Q_i, \sigma\rangle$ where $r \in P'$ is a rule of the form (1), σ is a substitution such that $F = B_i\sigma$, and $Q_i = B_1^{\prec} \wedge \ldots B_{i-1}^{\prec} \wedge B_{i+1}^{\preceq} \wedge \cdots \wedge B_k^{\preceq}$.

We described in [12] how to efficiently implement all of these operations apart from $T.\mathsf{mark}(F)$. For the latter, we associate with each fact a status bit, which we update lock-free using compare-and-set operations [6].

We use a queue C of resources: $C.\mathsf{enqueue}(c)$ atomically inserts a resource c into C; and $C.\mathsf{dequeue}$ atomically selects and removes a resource from c, or returns ε if no such resource exists. We also use a queue R of rules. One can implement these operations so that they are lock-free [6].

In addition to E, T, P, ρ, C, and R, we use several global variables: N is the number of threads (constant); W is the number of waiting threads (initially 0); P' is the 'current' program (initially P); run is a Boolean flag determining whether

Algorithm 2 evaluateUpdatedRules

1: $r := R.\mathsf{dequeue}$
2: **if** $r \neq \varepsilon$ **then**
3: **for** each $\tau \in T.\mathsf{evaluate}(\mathsf{b}(r)^{\preceq}, L, \emptyset)$ **do**
4: **if** $T.\mathsf{add}(\mathsf{h}(r)\tau)$ **then** notify all waiting threads
5: **return** $r \neq \varepsilon$

Algorithm 3 rewriteFacts

1: $c := C.\mathsf{dequeue}$
2: **if** $c \neq \varepsilon$ **then**
3: **for** each unmarked fact $F \in T$ that contains c **do**
4: **if** $T.\mathsf{mark}(F)$ and $T.\mathsf{add}(\rho(F))$ **then** notify all waiting threads
5: **return** $c \neq \varepsilon$

materialisation should continue (initially true); L is the last fact returned by T.next before P' is updated (initially undefined); and m is a mutex variable.

After initialising T to E, each of the N threads executes Algorithm 1, trying in line 2 to evaluate a rule whose resources have been updated, rewrite facts containing an outdated resource, or apply rules to a fact from T. When no work is available, the thread enters a critical section (lines 4–16) and waits for more work or a termination signal (line 5). Variable W is incremented (line 3) before entering, and decremented (line 15) after leaving the critical section, so at any point in time W is the number of threads inside the critical section. The thread goes to sleep (line 14) if no more work is available but other threads are running. When the last thread runs out of work (line 6), it adds to R an updated version of each outdated rule in P' (lines 7–8), notes the last fact in T (line 9), updates P' (line 10), signals termination if there are no rules to reevaluate (line 11), and wakes up all waiting threads (line 12). Updating P on a single thread simplifies the implementation, but it introduces a potential sequential bottleneck; however, our experiments have shown that, on most data sets, the amount of sequential processing in lines 7–12 does not significantly affect our approach.

Algorithm 2 processes the updated rules in R by evaluating their bodies in $T^{\preceq L}$ and instantiating the rule heads. Algorithm 3 rewrites all facts in T that contain an outdated resource c. Algorithm 4 extracts from T (line 1) an unprocessed, unmarked fact F and processes it. Fact F is rewritten if it is outdated (lines 4–5); this is needed because a thread can derive a fact containing an outdated resource *after* that resource has been processed by Algorithm 3. If F is an owl:sameAs triple with distinct resources (lines 6–7), then the smaller resource (according to an arbitrary total order) is selected as the representative for the other one, and the latter is added to the queue C of outdated resources (line 9). An ordering on resources is needed to prevent cyclic merges and to ensure uniqueness of the algorithm's result. The thread derives a contradiction if F is an owl:differentFrom triple with the same resource (lines 10–11). Otherwise, the thread applies the rules to F (lines 13–14) and derives the reflexive owl:sameAs

Algorithm 4 applyRules

1: $F := T.\mathsf{next}$
2: **if** $F \neq \varepsilon$ and F is not marked as outdated **then**
3: $\quad G := \rho(F)$
4: $\quad$ **if** $F \neq G$ **then**
5: $\quad\quad$ **if** $T.\mathsf{mark}(F)$ and $T.\mathsf{add}(G)$ **then** notify all waiting threads
6: $\quad$ **else if** F is of the form $\langle a, \mathsf{owl{:}sameAs}, b\rangle$ **then**
7: $\quad\quad$ **if** a and b are distinct **then**
8: $\quad\quad\quad c := \min\{a, b\}; \quad d := \max\{a, b\}$
9: $\quad\quad\quad$ **if** $\rho.\mathsf{mergeInto}(d, c)$ **then** $C.\mathsf{enqueue}(d)$ and notify all waiting threads
10: $\quad$ **else if** F is of the form $\langle a, \mathsf{owl{:}differentFrom}, a\rangle$ **then**
11: $\quad\quad$ derive a contradiction and notify all waiting threads
12: $\quad$ **else**
13: $\quad\quad$ **for** each $\langle r, Q, \sigma\rangle \in P'.\mathsf{rules}(F)$ and each $\tau \in T.\mathsf{evaluate}(Q, F, \sigma)$ **do**
14: $\quad\quad\quad$ **if** $T.\mathsf{add}(\mathsf{h}(r)\tau)$ **then** notify all waiting threads
15: $\quad\quad$ **for** each resource c occurring in F **do**
16: $\quad\quad\quad$ **if** $T.\mathsf{add}(\langle c, \mathsf{owl{:}sameAs}, c\rangle)$ **then** notify all waiting threads
17: **return** $F \neq \varepsilon$

triples (lines 15–16). Theorem 1 shows that our algorithm is correct, and its proof is given in Appendix A.

Theorem 1. *Algorithm 1 terminates for each finite set of facts E and program P. Let ρ be the final mapping and let T be the final set of unmarked facts.*

1. *$\langle a, \mathsf{owl{:}sameAs}, b\rangle \in T$ implies $a = b$—that is, ρ captures all equalities.*
2. *$F \in T$ implies $\rho(F) = F$—that is, T is minimal.*
3. *$T^\rho = [P \cup P_\approx]^\infty(E)$—that is, T and ρ represent $[P \cup P_\approx]^\infty(E)$.*
4. *Each pair of r and τ is considered at most once either in line 3 of Algorithm 2 or in line 13 of Algorithm 4—that is, derivations are not repeated.*

4.3 Implementing the Map of Representatives

We implemented the map ρ using two arrays, rep_ρ and next_ρ, indexed by resource IDs and initialised with zeros. Let c be a resource. Then, $\mathsf{rep}_\rho[c]$ is zero if c represents itself, or $\mathsf{rep}_\rho[c]$ contains a resource that c has been merged into. To retrieve resources equal to a representative, we organise each owl:sameAs-clique into a linked list of resources, so $\mathsf{next}_\rho[c]$ contains the next pointer.

Algorithm 5 merges d into c in a lock-free way. We update $\mathsf{rep}_\rho[d]$ to c if d currently represents itself (line 1). The *compare-and-set* primitive prevents thread interference: $\mathsf{CAS}(loc, exp, new)$ atomically loads the value stored at location loc into a temporary variable old, stores new into loc if $old = exp$, and returns old. If this update is successful, we append the clique of d to the clique of c (lines 2–4): we move to the end of c's list (short-circuit evaluation ensures that CAS in line 3 is evaluated only if $\mathsf{next}_\rho[e] = 0$), and then attempt to change $\mathsf{next}_\rho[e]$ to d; if the latter fails due to concurrent updates, we continue scanning c's list.

Algorithm 5 $\rho.\mathsf{mergeInto}(d, c)$

```
1: if CAS(rep_ρ[d], 0, c) = 0 then
2:     e := c
3:     while next_ρ[e] ≠ 0 or CAS(next_ρ[e], 0, d) ≠ 0 do
4:         e := next_ρ[e]
5:     return true
6: else
7:     return false
```

Algorithm 6 $\rho(c)$

```
1: r := c
2: loop
3:     r' := rep_ρ[r]
4:     if r' = 0 then
5:         return r
6:     else
7:         r := r'
```

Algorithm 6 computes $\rho(c)$ by traversing rep_ρ until it reaches a non-merged resource r. If another thread updates ρ by modifying $\mathsf{rep}_\rho[r]$, we just continue scanning rep_ρ past r, so the result is at least as current as ρ just before the start.

4.4 Example

Table 1 shows six steps in an application of our algorithm to the example program P_{ex} from Section 3 on one thread. Some resource names have been abbreviated for convenience, and $\approx$ abbreviates owl:sameAs. The triangle symbol identifies the last fact returned by T.next. Facts are numbered for easier referencing, and their (re)derivation is indicated on the right: $R(n)$ or $S(n)$ means that the fact was obtained from fact n and rule R or S; moreover, we rewrite facts right after merging resources, so $W(n)$ identifies a rewritten version of fact n, and $M(n)$ means that a fact was marked outdated because fact n caused ρ to change.

We start by extracting facts from T (Algorithm 4) and, in steps 1 and 2, we apply rule R to facts 2 and 3 to derive facts 4 and 5, respectively (lines 13–14 of Algorithm 4). In step 3, we extract fact 4, merge :America into :USA, mark facts 2 and 4 as outdated, and add their rewriting, facts 6 and 7, to T. In step 4 we merge :USA into :US, after which there are no further facts to process. Mapping ρ, however, has changed, so in lines 6–12 of Algorithm 1 we update P' to contain rules (R') and (S'), and add them to the queue R.

$$\langle x, \mathsf{owl{:}sameAs}, \mathsf{:US}\rangle \leftarrow \langle \mathsf{:Obama}, \mathsf{:presidentOf}, x\rangle \qquad (R')$$

$$\langle x, \mathsf{owl{:}sameAs}, \mathsf{:Obama}\rangle \leftarrow \langle x, \mathsf{:presidentOf}, \mathsf{:US}\rangle \qquad (S')$$

In step 5 we evaluate the rules in queue R (Algorithm 2), which introduces facts 9 and 10. Finally, in step 6, we merge :USPresident into :Obama and mark facts 1 and 9 as outdated. There is no more work at this point and the algorithm terminates, making only six derivations in total, instead of more than 60 derivations when owl:sameAs is axiomatised explicitly (see Section 3).

5 Answering SPARQL Queries over Rewritten Facts

Given a set of unmarked facts T and mapping ρ obtained from Algorithm 1, we can answer a SPARQL query Q correctly by evaluating Q in the expansion

Table 1. An Example Run of Algorithm 1 on P_{ex} and One Thread

Step 1		Step 2		Step 3	
1 ⟨:USPres, :presOf, :US⟩		1 ⟨:USPres, :presOf, :US⟩		1 ⟨:USPres, :presOf, :US⟩	
▷2 ⟨:Obama, :presOf, :Am⟩		2 ⟨:Obama, :presOf, :Am⟩		2 ~~⟨:Obama, :presOf, :Am⟩~~	$M(4)$
3 ⟨:Obama, :presOf, :US⟩		▷3 ⟨:Obama, :presOf, :US⟩		3 ⟨:Obama, :presOf, :US⟩	
4 ⟨:Am, ≈, :USA⟩	$R(2)$	4 ⟨:Am, ≈, :USA⟩		▷4 ~~⟨:Am, ≈, :USA⟩~~	$M(4)$
		5 ⟨:US, ≈, :USA⟩	$R(3)$	5 ⟨:US, ≈, :USA⟩	
				6 ⟨:Obama, :presOf, :USA⟩	$W(2)$
				7 ⟨:USA, ≈, :USA⟩	$W(4)$

Step 4		Step 5		Step 6	
1 ⟨:USPres, :presOf, :US⟩		1 ⟨:USPres, :presOf, :US⟩		1 ~~⟨:USPres, :presOf, :US⟩~~	$M(9)$
3 ⟨:Obama, :presOf, :US⟩	$W(6)$	3 ⟨:Obama, :presOf, :US⟩		3 ⟨:Obama, :presOf, :US⟩	
▷5 ~~⟨:US, ≈, :USA⟩~~	$M(5)$	8 ⟨:US, ≈, :US⟩	$R'(3)$	8 ⟨:US, ≈, :US⟩	
6 ~~⟨:Obama, :presOf, :USA⟩~~	$M(5)$	9 ⟨:USPres, ≈, :Obama⟩	$S'(1)$	▷9 ~~⟨:USPres, ≈, :Obama⟩~~	$M(9)$
7 ~~⟨:USA, ≈, :USA⟩~~	$M(5)$	10 ⟨:Obama, ≈, :Obama⟩	$S'(3)$	10 ⟨:Obama, ≈, :Obama⟩	$W(9)$
8 ⟨:US, ≈, :US⟩	$W(5,7)$				

T^ρ, but that is clearly inefficient. An attempt at an improvement would be to evaluate $\rho(Q)$ in T and expand the result—that is, for each answer μ obtained by evaluating $\rho(Q)$ in T, output each answer ν such that $\rho(\nu(x)) = \mu(x)$ for each variable x in the domain of μ. However, using the example program P_{ex} from Section 3, we show that such an approach is *not* complete. Note that, after we finish the materialisation of P_{ex}, we have $\rho(x) = \text{:US}$ for each $x \in \{\text{:USA}, \text{:AM}, \text{:US}\}$ and $\rho(x) = \text{:Obama}$ for each $x \in \{\text{:USPresident}, \text{:Obama}\}$.

The first problem is due to the *bag* semantics of SPARQL, where repeated answers matter. For example, let Q_1 be the following query:

$$\text{SELECT } ?x \text{ WHERE } \{\ ?x \text{ :presidentOf } ?y\ \} \qquad (Q_1)$$

On T^ρ, query Q_1 yields $\mu_1 = \{?x \mapsto \text{:Obama}\}$ and $\mu_2 = \{?x \mapsto \text{:USPresident}\}$, each repeated three times—once for each match of $?y$ to :USA, :US, or :America. In contrast, on T, query $\rho(Q_1)$ yields only one occurrence of μ_1, and its expansion produces one occurrence of μ_2; we thus obtain all answers, but not with the correct cardinalities. This problem arises because variable $?y$ is projected out, so the final expansion step does not take into account the number of times each binding of $?y$ contributes to the result. To solve this problem, we must modify the projection operator to output each projected answer as many times as there are resources in the projected owl:sameAs-clique(s). Thus, we can answer Q_1 as follows: we match the triple pattern of $\rho(Q_1)$ to T as usual, obtaining one answer $\nu_1 = \{?x \mapsto \text{:Obama}, ?y \mapsto \text{:US}\}$; then, we project $?y$ from ν_1 to obtain three occurrences of μ_1 since the owl:sameAs-clique of :US is of size three; finally, we expand each occurrence of μ_1 to μ_2 to obtain all six results.

The second problem is due to SPARQL builtin functions. For example, let Q_2 be the following SPARQL query:

$$\text{SELECT } ?y \text{ WHERE } \{\ ?x \text{ :presidentOf :US . BIND(STR}(?x) \text{ AS } ?y)\ \} \qquad (Q_2)$$

On T^ρ, query Q_2 yields $\tau_1 = \{?y \mapsto \text{“}\mathit{Obama}\text{”}\}$ and $\tau_2 = \{?y \mapsto \text{“}\mathit{USPresident}\text{”}\}$; in contrast, on T, query $\rho(Q_2)$ yields only τ_1, which does not expand into τ_2

because strings "Obama" and "USPresident" are not equal. To solve this problem, we must expand answers *before* evaluating builtin functions. Thus, we can answer Q_1 as follows: we match the triple pattern of $\rho(Q_2)$ to T as usual, obtaining $\kappa_1 = \{?x \mapsto \text{:Obama}\}$; then, we expand κ_1 to $\kappa_2 = \{?x \mapsto \text{:USPresident}\}$; next, we evaluate the BIND expression and extend κ_1 and κ_2 with the respective values for $?y$; finally, we project $?x$ to obtain τ_1 and τ_2. Since we have already expanded $?x$, we should not repeat the projected answers as many times as there are elements in the owl:sameAs-clique for $?x$; instead, we output each projected answer only once to obtain the correct cardinalities of answers.

6 Evaluation

We have implemented our approach in RDFox, and we have added an option to choose between handling owl:sameAs via rewriting (REW) or using the axiomatisation (AX) from Section 3. We then compared the performance of materialisation using these two approaches. In particular, we investigated the scalability of each approach with the number of threads, and we measured the effect that rewriting has on the number of derivations and materialised triples. All test data sets and the version of RDFox used are available online.[6]

Test Data Sets. We used five test data sets, each consisting of an OWL 2 DL ontology and a set of facts. The data sets were chosen because they contain axioms with the owl:sameAs property leading to interesting inferences. The following four data sets were obtained from real-world applications.

- Claros has been developed in an international collaboration between IT experts and archaeology and classical art research institutions with the aim of integrating disparate cultural heritage databases.[7]
- DBpedia is a crowd-sourced community effort to extract structured information from Wikipedia and make this information available on the Web.[8]
- OpenCyc is an extensive ontology about general human knowledge. It contains hundreds of thousands of terms organised in a carefully designed ontology and can be used as the basis of a wide variety of intelligent applications.[9]
- UniProt is a subset of an extensive knowledge base about protein sequences and functional information.[10]

The ontologies of all data sets other than DBpedia are not in the OWL 2 RL profile, so we first discarded all axioms outside OWL 2 RL, and then we translated the remaining axioms into datalog rules as described in [4]. Although RDFox supports arbitrary rules, and can thus handle the fixed OWL 2 RL/RDF rule set from [11, Section 4.3], reasoning with such rules is generally less efficient due to more complex joins in rule bodies.

[6] http://www.cs.ox.ac.uk/isg/tools/RDFox/ISWC-2014/
[7] http://www.clarosnet.org/XDB/ASP/clarosHome/
[8] http://www.dbpedia.org/
[9] http://www.cyc.com/platform/opencyc/
[10] http://www.uniprot.org/

Table 2. Test Data Sets Before and After Materialisation

	Rules	sA-rules	Triples before	Mode	Triples after unmarked	Triples after total	Memory (GB)	Rule appl.	Derivations	Merged resources
Claros	1312	42	19M	AX	102M		4.5	867M	11,009M	
				REW	79.5M	79.7M	3.6	149M	128M	12,890
				factor		1.28x	1.28x	5.8x	85.5x	
DBPedia	3384	23	113M	AX	139M		6.9	934M	895M	
				REW	136M	136M	7.0	44.5M	37M	7,430
				factor		1.2x	0.99x	21.0x	24.4x	
OpenCyc	261,067	3,781	2.4M	AX	1,176M		35.9	7,832M	12,890M	
				REW	141M	142M	4.6	309M	281M	361,386
				factor		7.8x	7.8x	25.3x	45.9x	
UniProt	451	60	123M	AX	228M		15.1	1,801M	1,555M	
				REW	228M	228M	15.1	262M	183M	5
				factor		1.0x	1.0x	6.9x	8.5x	
UOBM	279	4	2.2M	AX	36M		1.2	332M	16,152M	
				REW	9.4M	9.7M	0.4	33.8M	4,256M	686
				factor		3.2x	3.2x	9.9x	3.8x	

Our fifth data set was UOBM [10]—a synthetic data set that extends the well-known LUBM [5] benchmark. We did not use LUBM because neither its ontology nor its data uses the owl:sameAs property. The UOBM ontology is also outside OWL 2 RL; however, instead of using its OWL 2 RL subset, we used its *upper bound* [18]—an unsound but complete OWL 2 RL approximation of the original ontology; thus, all answers that can be obtained from the original ontology can also be obtained from the upper bound, but not the other way around. Efficient materialisation of the upper bound was critical for the work presented in [18], and it has proved to be challenging due to equality reasoning.

The left-hand part of Table 2 summarises our test data sets: column 'Rules' shows the total number of rules, column 'sA-rules' shows the number of rules containing the owl:sameAs property in the head, and column 'Triples before' shows the number of triples before materialisation.

Test Setting. We conducted our tests on a Dell computer with 128 GB of RAM and two Xeon E5-2643 processors with a total of 8 physical and 16 virtual cores, running 64-bit Fedora release 20, kernel version 3.13.3-201. We have not conducted warm and cold start tests separately since, as a main-memory system, the performance of RDFox should not be affected by the state of the operating system's buffers. For the AX tests, we extended the relevant datalog program with the seven rules from Section 3. We verified that the expansion of the rewritten triples is identical to the triples derived using the axiomatisation.

Effect of Rewriting on Total Work. In order to see how rewriting affects the total amount of work, we materialised each test data set in both AX and REW modes while collecting statistics about the inference process; the results are shown in the right-hand part of Table 2. Column 'Triples after' shows the number of triples after materialisation; in the case of REW tests, we additionally show the number of unmarked triples (i.e., of triples relevant to query answering). Column 'Memory' shows the total memory use as measured by RDFox's internal counters. Column 'Rule appl.' shows the total number of times a rule has been

Table 3. Materialisation Times with Axiomatisation and Rewriting

Test	Claros					DBpedia					OpenCyc				
Threads	AX		REW		$\frac{\text{AX}}{\text{REW}}$	AX		REW		$\frac{\text{AX}}{\text{REW}}$	AX		REW		$\frac{\text{AX}}{\text{REW}}$
	sec	spd	sec	spd		sec	spd	sec	spd		sec	spd	sec	spd	
1	2042.9	1.0	65.8	1.0	31.1	219.8	1.0	31.7	1.0	6.9	2093.7	1.0	119.9	1.0	17.5
2	969.7	2.1	35.2	1.9	27.6	114.6	1.9	17.6	1.8	6.5	1326.5	1.6	78.3	1.5	16.9
4	462.0	4.4	18.1	3.6	25.5	66.3	3.3	10.7	3.0	6.2	692.6	3.0	40.5	3.0	17.1
8	237.2	8.6	9.9	6.7	24.1	36.1	6.1	5.2	6.0	6.9	351.3	6.0	23.0	5.2	15.2
12	184.9	11.1	7.9	8.3	23.3	31.9	6.9	4.1	7.7	7.7	291.8	7.2	56.2	2.1	5.5
16	153.4	13.3	6.9	9.6	22.3	27.5	8.0	3.6	8.8	7.7	254.0	8.2	52.3	2.3	4.9

Test	UniProt					UOBM				
Threads	AX		REW		$\frac{\text{AX}}{\text{REW}}$	AX		REW		$\frac{\text{AX}}{\text{REW}}$
	sec	spd	sec	spd		sec	spd	sec	spd	
1	370.6	1.0	143.4	1.0	2.6	2696.7	1.0	1152.7	1.0	2.3
2	232.3	1.6	86.7	1.7	2.7	1524.6	1.8	599.6	1.9	2.5
4	129.2	2.9	46.5	3.1	2.8	813.3	3.3	318.3	3.6	2.6
8	74.7	5.0	25.1	5.7	3.0	439.9	6.1	177.7	6.5	2.5
12	61.0	6.1	19.9	7.2	3.1	348.9	7.7	152.7	7.6	2.3
16	61.9	6.0	17.1	8.4	3.6	314.4	8.6	137.9	8.4	2.3

applied to a triple (i.e., the number of times a rule is considered in line 13 of Algorithm 4), and column 'Derivations' shows the total number of derivations (i.e., the number of times a substitution is considered in line 14 of Algorithm 4). Column 'Merged resources' shows the number of resources that were replaced with representatives in the course of materialisation. Finally, the row 'factor' shows the ratio between the respective values in the AX and the REW tests.

As one can see from the table, the reduction in the number of the derived triples is correlated with the number of normalised constants: on UniProt there is no observable reduction since only five resources are merged; however, equalities proliferate on OpenCyc and so rewriting is particularly effective. In all cases the difference between the numbers of all and unmarked triples is negligible, suggesting that our decision to mark, rather than delete triples does not have unexpected drawbacks. In all cases, however, the reduction in the number of rule applications and, in particular, of derivations is much more pronounced than the reduction in the number of derived triples.

Effect of Rewriting on Materialisation Times. In order to see how rewriting affects materialisation times, we measured the wall-clock times needed to materialise our test data sets in AX and REW modes on on 1, 2, 4, 8, 12, and 16 threads. For each test, we report average wall-clock time over three runs. Table 3 shows our test results. For each test, column 'sec' shows the materialisation time in seconds, column 'spd' shows the speedup over the single-threaded version, and column '$\frac{\text{AX}}{\text{REW}}$' shows the speedup of REW over AX.

As one can see from the table, RDFox parallelises computation exceptionally well in both AX and REW modes. When using the eight physical cores of our test server, the speedup is consistently between six and seven, which suggests that the lock-free algorithms and data structures of RDFox are very effective. We do not know exactly what caused more-than-linear speedup on Claros: one possibility is that, due to chance, different threads accessed memory more locally thus incurring fewer CPU cache misses. The speedup continues to increase with

hyperthreading, but it is less pronounced: virtual cores do not provide additional execution resources, and so they mainly compensate for CPU stalls due to cache misses. The AX mode seems to scale better with the number of threads than the REW mode, and we believe this to be due to contention between threads while accessing the map ρ. Only OpenCyc in REW mode did not scale particularly well, which can be explained in two ways: the materialisation of OpenCyc produces a large number of cliques (see Table 2), so there might be some contention in accessing and maintaining ρ; furthermore, OpenCyc contains many rules, so the sequential updating of the program in lines 7–10 of Algorithm 1 becomes significant. Finally, since the materialisation of Claros with more than eight threads in REW mode takes less than ten seconds, these results are difficult to measure and are susceptible to skew.

Our results show that rewriting can significantly reduce materialisation times. RDFox was consistently faster in the REW mode than in the AX mode even on UniProt, where, as one can see in Table 2, the reduction in the number of triples is negligible. This seems to be due to the reduction in the number of derivations, mainly involving rules EQ_1–EQ_5, which is still significant on UniProt. In all cases, the speedup of rewriting is typically much larger than the reduction in the number of derived triples (cf. Table 2), suggesting that the primary benefit of rewriting lies in reducing the work needed to match the rules, and not in reducing the number of derived triples. This observation is consistent with the fact that the speedup of rewriting was not so pronounced on UniProt and UOBM: the reduction in the number of derivations on these tests was less significant.

Our analysis of the derivations that RDFox makes on UOBM revealed that, due to the derived owl:sameAs triples, the materialisation contains large numbers of resources connected by the :hasSameHomeTownWith property. This property is also symmetric and transitive so, for each pair of connected resources, the number of times each triple is derived by the transitivity rule is quadratic in the number of connected resources. This leads to a large number of redundant derivations that do *not* involve equality. Thus, although it is helpful, rewriting does not reduce the number of derivation in the same way as, for example, on Claros, which explains the relatively modest speedup of REW over AX.

7 Conclusion

In this paper we have presented a novel approach to the materialisation of datalog programs in centralised, main-memory, multi-core systems that handle the owl:sameAs property via rewriting—that is, by replacing equal resources with a single representative. We have shown empirically that our approach can reduce reasoning times on practical data sets by orders of magnitude. In our future work we aim to extend the capabilities of RDFox in several directions. These will include supporting incremental reasoning, with the main challenge being the interaction with owl:sameAs rewriting, and increasing storage capacity by using a distributed, shared-nothing system, with the main challenge being to exploit the structure of RDF data so as to minimise distributed processing.

Acknowledgements

This work was supported by the EPSRC projects ExODA and MaSI^3.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1995)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: Owlim: A family of scalable semantic repositories. Semantic Web 2(1), 33–42 (2011)
4. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description Logic Programs: Combining Logic Programs with Description Logic. In: Proc. WWW. pp. 48–57 (2003)
5. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics 3(2–3), 158–182 (2005)
6. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
7. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission (May 21 2004)
8. Kolovski, V., Wu, Z., Eadon, G.: Optimizing Enterprise-Scale OWL 2 RL Reasoning in a Relational Database System. In: Proc. ISWC. pp. 436–452 (2010)
9. Larson, P.A.: Letter from the Special Issue Editor. IEEE Data Engineering Bulletin, Special Issue on Main-Memory Database Systems 36(2), 5 (2013)
10. Ma, L., Yang, Y., Qiu, Z., Xie, G.T., Pan, Y., Liu, S.: Towards a Complete OWL Ontology Benchmark. In: Proc. ESWC. pp. 125–139 (2006)
11. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language: Profiles, W3C Recommendation (October 27 2009)
12. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In: Proc. AAAI (2014)
13. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 7, pp. 371–443. Elsevier Science (2001)
14. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. Journal of Web Semantics 7(4), 305–316 (2009)
15. Stocker, M., Smith, M.: Owlgres: A Scalable OWL Reasoner. In: Proc. OWLED: Experiences and Directions Workshop. pp. 26–27 (2008)
16. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.E.: WebPIE: A Web-scale Parallel Inference Engine using MapReduce. Journal of Web Semantics 10, 59–75 (2012)
17. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In: Proc. ICDE. pp. 1239–1248 (2008)
18. Zhou, Y., Cuenca Grau, B., Horrocks, I., Wu, Z., Banerjee, J.: Making the most of your triple store: query answering in OWL 2 using an RL reasoner. In: Proc. WWW. pp. 1569–1580 (2013)

A Proof of Theorem 1

Let S be a finite set of facts. For r a rule of the form (1), $r(S)$ is the smallest set such that $H\sigma \in r(S)$ for each substitution σ satisfying $B_i\sigma \in S$ for each i with $1 \leq i \leq n$; moreover, for P a datalog program, let $P(S) := \bigcup_{r \in P} r(S)$.

Given a datalog program P and a finite set of explicit facts E, let $P^0(E) := E$; let $P^i(E) := P^{i-1}(E) \cup P(P^{i-1}(E))$ for each $i > 0$; and let $P^\infty(E) := \bigcup_i P^i(E)$.

Theorem 1. *Algorithm 1 terminates for each finite set of facts E and program P. Let ρ be the final mapping and let T be the final set of unmarked facts.*

1. *$\langle a, \mathsf{owl{:}sameAs}, b\rangle \in T$ implies $a = b$—that is, ρ captures all equalities.*
2. *$F \in T$ implies $\rho(F) = F$—that is, T is minimal.*
3. *$T^\rho = [P \cup P_\approx]^\infty(E)$—that is, T and ρ represent $[P \cup P_\approx]^\infty(E)$.*
4. *Each pair of r and τ is considered at most once either in line 3 of Algorithm 2 or in line 13 of Algorithm 4—that is, derivations are not repeated.*

For convenience, let $\Pi^i := [P \cup P_\approx]^i(E)$ and let $\Pi^\infty := [P \cup P_\approx]^\infty(E)$. Furthermore, let N_r be the number of distinct resources occurring in E, and let $|P|$ be the number of rules in P. We split our proof into several distinct claims.

Claim. The algorithm terminates.

Proof. Duplicate facts are eliminated eagerly, and facts are never deleted, so the number of successful additions to T is bounded by N_r^3. Moreover, Algorithm 5 ensures that each resource is merged at most once; hence, ρ can change at most N_r times, and the number of additions to queue C is bounded by N_r as well. Thus, $P' \neq \rho(P')$ may fail in lines lines 6–12 of Algorithm 1 at most N_r times, so the number of additions to queue R is bounded by $|P| \cdot N_r$. Together, these observations clearly imply that the algorithm terminates. □

All operations used in our algorithm are linearisable and the algorithm terminates, so the execution of N threads on input E and P has the same effect as some finite sequence $\Lambda = \langle \lambda_1, \ldots, \lambda_\ell \rangle$ of operations where each λ_i is

- $T.\mathsf{add}(F)$, representing successful addition of F to T in line 4 of Algorithm 2, line 4 of Algorithm 3, or line 5, 14, or 16 of Algorithm 4,
- $F := T.\mathsf{next}$, representing successful extraction of an unmarked, unprocessed fact F from T in line 1 of Algorithm 4,
- $T.\mathsf{mark}(F)$, representing successful marking of F as outdated in line 4 of Algorithm 3 or line 5 of Algorithm 4,
- $\rho.\mathsf{mergeInto}(d, c)$, representing successful merging of resource d into resource c in line 9 of Algorithm 4, or
- $P' := \rho(P)$, representing an update of program P' in line 10 of Algorithm 1.

Materialisation first adds all facts in E to T, so the first m operations in Λ are of the form $T.\mathsf{add}(F_i)$ for each $F_i \in E$. By a slight abuse of notation, we often treat Λ as a set and write $\lambda_i \in \Lambda$. Our algorithm clearly ensures that each operation

$T.\mathsf{add}(F)$ in Λ is followed in Λ by $F := T.\mathsf{next}$ or $T.\mathsf{mark}(F)$; if both of these operations occur in Λ, then the former precedes the latter. Sequence Λ induces a sequence of mappings of resources to representatives $\rho_0, \ldots, \rho_\ell$: mapping ρ_0 is identity; for each $i > 0$ with $\lambda_i = \rho.\mathsf{mergeInto}(d, c)$, mapping ρ_i is obtained from ρ_{i-1} by setting $\rho_i(a) := \rho_{i-1}(c)$ for each resource a with $\rho_{i-1}(a) = d$; and for each $i > 0$ with $\lambda_i \neq \rho.\mathsf{mergeInto}(d, c)$, let $\rho_i := \rho_{i-1}$. Clearly, $\rho = \rho_\ell$; furthermore, for each i with $1 \leq i \leq \ell$, if $\rho_i(F) \neq F$, then $\rho_j(F) \neq F$ for each j with $i \leq j \leq \ell$.

Claim. $\langle a, \mathsf{owl{:}sameAs}, b\rangle \in T$ implies $a = b$.

Proof. Assume that T contains an unmarked fact $F = \langle a, \mathsf{owl{:}sameAs}, b\rangle$ with $a \neq b$. Then, there exists an operation $\lambda_i \in \Lambda$ of the form $F := T.\mathsf{next}$. In case $\rho_i(F) \neq F$, then lines 4–5 of Algorithm 4 ensure that $T.\mathsf{mark}(F) \in \Lambda$, contradicting the assumption that F was unmarked. In case $\rho_i(F) = F$, then there exists an operation $\lambda_j \in \Lambda$ with $j \geq i$ of the form $\rho.\mathsf{mergeInto}(a, b)$ or $\rho.\mathsf{mergeInto}(b, a)$. Thus, either a or b is added to queue C in line 9 of Algorithm 4, and this resource is later processed in Algorithm 3; then, due to line 4 of Algorithm 3, there exists an operation $\lambda_k \in \Lambda$ with $k \geq j$ of the form $T.\mathsf{mark}(F)$. We obtain a contradiction in either case, as required. □

Claim. $F \in T$ implies $\rho(F) = F$.

Proof. Assume for the sake of contradiction that a fact $F \in T$ exists such that $F \neq \rho(F)$. Then, there exists an operation $\lambda_i \in \Lambda$ of the form $F := T.\mathsf{next}$. We clearly have $F = \rho_i(F)$, or lines 4–5 of Algorithm 4 would have ensured that $T.\mathsf{mark}(F) \in \Lambda$. But then, there exists an operation $\lambda_j \in \Lambda$ with $i < j \leq \ell$ of the form $\rho.\mathsf{mergeInto}(d, c)$ where resource d occurs in F. Hence, d is added to queue C in line 9 of Algorithm 4, and this resource is later processed in Algorithm 3; but then, there exists an operation $\lambda_k \in \Lambda$ with $k \geq j$ of the form $T.\mathsf{mark}(F)$, which contradicts our assumption that $F \in T$. □

Claim. $T^\rho \subseteq \Pi^\infty$.

Proof. The claim holds because $P_\approx$ contains replacement rules (EQ_2)–(EQ_4) and the following two properties are satisfied for each $1 \leq i \leq \ell$:

(i) for each resource a, we have $\langle a, \mathsf{owl{:}sameAs}, \rho_i(a)\rangle \in \Pi^\infty$, and
(ii) if λ_i is of the form $T.\mathsf{add}(F)$, then $F \in \Pi^\infty$.

We prove (i) and (ii) by induction on i. For the induction base, we consider the first m operations in Λ of the form $T.\mathsf{add}(F_i)$ with $F_i \in E$; both claims clearly hold for $i = m$. For the inductive step, property (i) can be affected only if λ_i is of the form $\rho.\mathsf{mergeInto}(d, c)$; furthermore, property (ii) can be affected only if λ_i is of the form $T.\mathsf{add}(F)$; hence, we analyse these cases separately.

Assume $\lambda_i = \rho.\mathsf{mergeInto}(d, c)$. To show that property (i) holds, consider an arbitrary resource a; property (i) holds trivially for a if $\rho_{i-1}(a) \neq d$, so assume that $\rho_{i-1}(a) = d$. Due to the form of λ_i, constant d was added to C in line 9 of Algorithm 4 due to an operation $\lambda_j \in \Lambda$ with $j < i$ of the form $F := T.\mathsf{next}$

with F of the form $\langle c, \mathsf{owl{:}sameAs}, d\rangle$ or $\langle d, \mathsf{owl{:}sameAs}, c\rangle$; but then, there exists an operation $\lambda_k \in \Lambda$ with $k < j$ of the form $T.\mathsf{add}(F)$; thus, by the induction assumption, property (ii) implies $F \in \Pi^\infty$. Furthermore, by the induction assumption and $\rho_{i-1}(d) = d$, we have $\langle a, \mathsf{owl{:}sameAs}, d\rangle \in \Pi^\infty$, and we also have $\langle c, \mathsf{owl{:}sameAs}, \rho_{i-1}(c)\rangle \in \Pi^\infty$. Moreover, $\rho_i(a) = \rho_{i-1}(c) = \rho_i(c)$ holds by Algorithm 5. Finally, property $\mathsf{owl{:}sameAs}$ is reflexive, symmetric, and transitive in Π^∞, so $\langle a, \mathsf{owl{:}sameAs}, \rho_i(a)\rangle \in \Pi^\infty$ holds, as required for property (i).

Assume $\lambda_i = T.\mathsf{add}(F)$ by line 4 of Algorithm 3 or line 5 of Algorithm 4; thus, F obtained from some fact G for which there exists an operation $\lambda_j \in \Lambda$ with $j < i$ of the form $T.\mathsf{add}(G)$. By the induction assumption, we have $G \in \Pi^\infty$. Fact G is obtained from F by replacing each occurrence of a resource c in F with $\rho_n(c)$ for some n with $1 \leq n \leq i$; by the induction assumption, mapping ρ_n satisfies property (i), so we have $\langle c, \mathsf{owl{:}sameAs}, \rho_n(c)\rangle \in \Pi^\infty$. But then, replacement rules (EQ_2)–(EQ_4) in $P_\approx$ ensure that $F \in \Pi^\infty$.

Assume $\lambda_i = T.\mathsf{add}(F)$ by line 4 of Algorithm 2 or line 14 of Algorithm 4; thus, F is obtained by applying a rule $\rho_i(r)$ of the form (1) via a substitution τ that matches the body atoms $B_1, \ldots, B_n$ of $\rho_i(r)$ to facts $F_1, \ldots, F_n$ such that, for each $1 \leq j \leq n$, sequence Λ contains an operation of the form $T.\mathsf{add}(F_j)$ preceding λ_i. By the induction assumption, property (ii) implies $\{F_1, \ldots, F_n\} \subseteq \Pi^\infty$. We next show that rule r can be applied to facts $\{F'_1, \ldots, F'_n\} \subseteq \Pi^\infty$ to derive a fact F', and that the rules in $P_\approx$ can be used to derive F from F'. Let $C_r = \{c_1, \ldots, c_k\}$ be the set of resources occurring in rule r; by the induction assumption, property (i) ensures the following observation:

$$\langle c_j, \mathsf{owl{:}sameAs}, \rho_i(c_j)\rangle \in \Pi^\infty \text{ for each } j \text{ with } 1 \leq j \leq k. \qquad (\Diamond)$$

Let $F'_1, \ldots, F'_n$ be the facts obtained from $F_1, \ldots, F_n$ by replacing, for each $c \in C_r$, each occurrence of $\rho_i(c)$ with c; due to $(\Diamond)$, the rules in $P_\approx$ ensure that $\{F'_1, \ldots, F'_n\} \subseteq \Pi^\infty$ holds. Now let σ be the substitution obtained from τ by replacing, for each $c_j \in C_r$, resource $\rho_i(c_j)$ in the range of τ with c_j; then, substitution σ matches all body atoms of r to derive fact $F' \in \Pi^\infty$ where $\rho_i(F') = F$. Due to $(\Diamond)$, the rules in $P_\approx$ ensure $F \in \Pi^\infty$, as required for property (ii).

Assume $\lambda_i = T.\mathsf{add}(F)$ by line 16 of Algorithm 4, so $F = \langle c, \mathsf{owl{:}sameAs}, c\rangle$ with c a resource occurring in some fact G for which there exists an operation $\lambda_j \in \Lambda$ with $j < i$ of the form $T.\mathsf{add}(G)$. By the induction assumption, we have $G \in \Pi^\infty$. But then, due to rules (EQ_1) in $P_\approx$, we have $F \in \Pi^\infty$, as required. $\square$

Before showing that $T^\rho \supseteq \Pi^\infty$, we prove a useful property $(\Diamond)$, which essentially says that, whenever a fact F is added to T in operation j, at each step i after j, a rewriting G of F is or will be 'visible' in T (i.e., G has not been marked outdated before operation i).

Claim $(\Diamond)$. For each i with $1 \leq i \leq \ell$ and each operation $\lambda_j \in \Lambda$ with $j \leq i$ of the form $T.\mathsf{add}(F)$, there exists k such that

(a) $\lambda_k = T.\mathsf{add}(G)$,

(b) G is obtained from F by replacing each occurrence of a resource c with $\rho_n(c)$ for some n with $n \leq k$, and
(c) for each k' (if any) with $k < k' \leq i$, we have $\lambda_{k'} \neq T.\mathsf{mark}(G)$.

Proof. The proof proceeds by induction on i. The base case $i = 0$ is vacuous, so we assume that ($\Diamond$) holds up to some i with $1 \leq i < \ell$, and we consider operation λ_{i+1} and an arbitrary operation λ_j with $j \leq i+1$ of the form $T.\mathsf{add}(F)$. If $j = i+1$, then the claim trivially holds for $k = j$; otherwise, we have $j < i+1$, so by applying the induction assumption to i, an integer k with $\lambda_k = T.\mathsf{add}(G)$ satisfying properties (a)–(c). If $\lambda_{i+1} \neq T.\mathsf{mark}(G)$, then k satisfies properties (a)–(c) for $i+1$ and λ_j as well. If, however, $\lambda_{i+1} = T.\mathsf{mark}(G)$, then either in line 4 of Algorithm 3 or in line 5 of Algorithm 4 an attempt will be made to add a fact G' satisfying property (b) to T. First, assume that G' already exists in T—that is, some $m \leq i$ exists such that $\lambda_m = T.\mathsf{add}(G')$; then, by applying the induction assumption to G', some k' with $\lambda_{k'} = T.\mathsf{add}(G'')$ exists that satisfies properties (a)–(c); but then, k' satisfies properties (a)–(c) for F which proves the claim for λ_j. In contrast, if no such m exists, then the addition in line 4 of Algorithm 3 or line 5 of Algorithm 4 succeeds, and some k' with $i+1 < k'$ exists such that $\lambda_{k'} = T.\mathsf{add}(G')$. Thus, properties (a)–(c) hold for $i+1$ and λ_j. □

Claim. $T^\rho \supseteq \Pi^\infty$.

Proof. The claim holds if $T \supseteq \rho(\Pi^\infty)$ and if $\langle c, \mathsf{owl{:}sameAs}, d\rangle \in \Pi^\infty$ implies $\rho(c) = \rho(d)$. Thus, we prove by induction on i that each set Π^i in the sequence $\Pi^0, \Pi^1, \ldots$ satisfies the following two properties:

(i) $T \supseteq \rho(\Pi^i)$ and
(ii) $\langle c, \mathsf{owl{:}sameAs}, d\rangle \in \Pi^i$ implies $\rho(c) = \rho(d)$.

To prove these claims, we next consider an arbitrary fact $F \in \Pi^0$ (for the base case) or $F \in \Pi^{i+1} \setminus \Pi^i$ with $i \geq 0$ (for the induction step), and we show that $T.\mathsf{add}(\rho(F)) \in \Lambda$. Since ρ is the final resource mapping, this fact will never be marked as outdated and so we have $\rho(F) \in T$, as required for property (i). Moreover, if $F = \langle c, \mathsf{owl{:}sameAs}, d\rangle$, then $\rho(F) \in T$ together with property (1) of Theorem 1 imply that $\rho(F)$ is of the form $\langle a, \mathsf{owl{:}sameAs}, a\rangle$, and so we have $\rho(c) = a = \rho(d)$, as required for property (ii).

Induction Base. Consider an arbitrary fact $F \in \Pi^0 = E$. Let G be the fact that satisfies (a)–(c) of property ($\Diamond$) for $i = \ell$; fact G is never marked as outdated due to (c), so $\rho(F) = G$ holds by property 2 of Theorem 1. But then, (a) implies $T.\mathsf{add}(\rho(F)) \in \Lambda$, as required.

Induction Step. Fact $F \in \Pi^{i+1} \setminus \Pi^i$ is derived using a rule $r \in P \cup P_\approx$ of the form (1) from facts $\{F_1, \ldots, F_n\} \subseteq \Pi^i$. By the induction assumption we have $\{\rho(F_1), \ldots, \rho(F_n)\} \subseteq T$, which implies $T.\mathsf{add}(\rho(F_j)) \in \Lambda$ for each $1 \leq j \leq n$; we denote the latter property with (†).

Assume that F is derived by applying rule (EQ_1) to a fact $G \in \Pi^i$. By property (†), there exists an operation $T.\mathsf{add}(\rho(G)) \in \Lambda$; thus, there exists an operation $\rho(G) := T.\mathsf{next} \in \Lambda$; finally, line 16 of Algorithm 4 ensures $T.\mathsf{add}(\rho(F)) \in \Lambda$.

Assume that F is derived by applying rule (EQ_2), (EQ_3), or (EQ_4) to facts $\{G, \langle c, \mathsf{owl{:}sameAs}, d\rangle\} \subseteq \Pi^i$. By the induction assumption, we have $\rho(c) = \rho(d)$. But then, since G is obtained from F by replacing c with d, we have $\rho(G) = \rho(F)$; by property (†), we have $T.\mathsf{add}(\rho(G)) \in \Lambda$, and so $T.\mathsf{add}(\rho(F)) \in \Lambda$.

Assume that F is derived by applying rule (EQ_5) to a fact $G \in \Pi^i$. By property (†), there exists an operation $T.\mathsf{add}(\rho(G)) \in \Lambda$; thus, there exists an operation $\rho(G) := T.\mathsf{next} \in \Lambda$; finally, line 11 of Algorithm 4 ensures $T.\mathsf{add}(\rho(F)) \in \Lambda$.

Assume that F is derived by applying a rule $r \in P$ of form (1) to facts $\{F_1, \ldots, F_n\} \subseteq \Pi^i$ via some substitution σ. Let τ be the substitution where $\tau(x) = \rho(\sigma(x))$ for each variable x from the domain of σ. Rule $\rho(r)$ derives $\rho(F)$ via τ from $\rho(F_1), \ldots, \rho(F_n)$. To show that one can match these facts to an annotated query derived from $\rho(r)$, let G be the fact among $\rho(F_1), \ldots, \rho(F_n)$ for which operation $T.\mathsf{add}(G)$ occurs last in Λ, and let j be the smallest integer with $1 \leq j \leq n$ and $\rho(F_j) = G$. Rule $\rho(r)$ occurs in the final program P', so we have two possibilities.

- Assume that $\rho(r) \notin P$ and that $\rho(r)$ occurs for the first time in Λ in an operation $P' := \rho(P)$ that appears in Λ after operation $T.\mathsf{add}(G)$. Then, by property (†), rule $\rho(r)$ is applied to facts $\rho(F_1), \ldots, \rho(F_n)$ in line 3 of Algorithm 2, and so $T.\mathsf{add}(\rho(F)) \in \Lambda$ due to line 4 of Algorithm 2.
- In all other cases, Λ contains an operation $\rho(F_j) := T.\mathsf{next}$, and at that point program P' contains $\rho(r)$; hence, rule $\rho(r)$ is applied in line 13 of Algorithm 4 by matching $\rho(B_j)$ to G. Now by (†) and the way in which we have selected G, we have $\{\rho(F_1), \ldots, \rho(F_n)\} \subseteq T^{\preceq G}$, so each body atom $\rho(B_k)$ of $\rho(r)$ can be matched to $T^{\preceq G}$. Furthermore, j is the smallest index such that $\rho(B_j) = G$, so $\rho(F_k) \in T^{\prec G}$ for each $1 \leq k < j$. Thus, substitution τ is returned in line 13 of Algorithm 4, and so line 14 of Algorithm 4 ensures that $T.\mathsf{add}(\rho(F)) \in \Lambda$ holds, as required. □

Claim. Each pair of r and τ is considered at most once either in line 3 of Algorithm 2 or in line 13 of Algorithm 4.

Proof. For the sake of contradiction, assume that a rule r of the form (1) and substitution τ exist that violate this claim. The domain of τ contains all variables in r, so τ matches all body atoms of r to a unique set of facts $F_1, \ldots, F_n$. We next show that the annotations in queries prevent the algorithm from considering the same r and τ more than once. To this end, let $G \in \{F_1, \ldots, F_n\}$ be the fact for which operation $T.\mathsf{add}(G)$ occurs last in Λ.

Assume that $T.\mathsf{add}(G)$ occurs in Λ before the operation $P' := \rho(P)$ in Λ with $r \in P'$ but $r \notin P$. Since P' is updated in line 10 of Algorithm 1 only when there are no facts to process, operation $G := T.\mathsf{next}$ also occurs in Λ before $P' := \rho(P)$; but then, r cannot be applied to G in line 13 of Algorithm 4. Hence, the only possibility is that r and τ are considered twice is in line 3 of Algorithm 2; however, line 7 of Algorithm 1 ensures that r is enqueued into R at most once.

Assume that $T.\mathsf{add}(G)$ occurs in Λ after the operation $P' := \rho(P) \in \Lambda$ with $r \in P'$. Line 3 of Algorithm 2 evaluates the rules in R only up to the last fact L extracted from T before P' is updated, and so r cannot be matched in G in

line 3 of Algorithm 2; hence, the only possibility is that r and τ are considered twice in line 13 of Algorithm 4. To this end, assume that F and F' are (not necessarily distinct) facts extracted in line 1 of Algorithm 4, let Q the annotated query used to match body atom B_i of r to F, and let Q' the annotated query used to match body atom B_j of r to F'; thus, we have $B_i\tau = F$ and $B_j\tau = F'$. We consider the following two cases.

- Assume $F = F'$; furthermore, w.l.o.g. assume that $i \leq j$. If $i = j$, we have a contradiction since operation $F := T.\mathsf{next}$ occurs in Λ only once and query $Q = Q'$ is considered in line 13 of Algorithm 4 only once. If $i < j$, we have a contradiction since $\bowtie_i = <$ holds in the annotated query Q', so atom B_i cannot be matched to fact F in query Q' (i.e., we cannot have $B_i\tau = F$) due to $F \notin T^{<F'} = T^{<F}$.
- Assume $F \neq F'$; furthermore, w.l.o.g. assume that operation $T.\mathsf{add}(F)$ occurs in Λ after operation $T.\mathsf{add}(F')$. But then, $B_j\tau = F'$ leads to a contradiction since atom B_j cannot be matched to fact F' in query Q when fact F is extracted in line 1 of Algorithm 4. $\square$