

Parallel OWL 2 RL Materialisation in Centralised, Main-Memory RDF Systems

Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks and Dan Olteanu
firstname.lastname@cs.ox.ac.uk

Department of Computer Science, Oxford University, Oxford, United Kingdom

Abstract. We present a novel approach to parallel *materialisation* (i.e., fixpoint computation) of OWL RL Knowledge Bases in centralised, main-memory, multi-core RDF systems. Our approach comprises a datalog reasoning algorithm that evenly distributes the workload to cores, and an RDF indexing data structure that supports efficient, ‘mostly’ lock-free parallel updates. Our empirical evaluation shows that our approach parallelises computation very well so, with 16 physical cores, materialisation can be up to 13.9 times faster than with just one core.

1 Introduction

The OWL 2 RL profile forms a fragment of datalog so that reasoning over OWL 2 RL knowledge bases (KB) can straightforwardly be rendered into answering datalog queries. These datalog queries are either the OWL 2 RL/RDF rules [18, Section 4.3] applied to the KB or the datalog translations [10] of the OWL 2 RL axioms of the KB applied to the data portion (ABox) of the KB. Answering datalog queries can be solved by backward chaining [2, 23], or one can *materialise* all consequences of the rules and the data so that subsequent queries can be answered without the rules. Materialisation supports efficient querying, so it is commonly used in practice, but it is also very expensive. We show that materialisation can be efficiently parallelised on modern multi-core systems. In addition, main-memory databases have been gaining momentum in academia and practice [16] due to the decreasing cost of RAM, so we focus on centralised, main-memory, multi-core RDF systems. We present a new materialisation algorithm that evenly distributes the workload to cores, and an RDF data indexing scheme that supports efficient ‘mostly’ lock-free data insertion. Our techniques are complementary to the ones for shared-nothing distributed RDF systems with nontrivial communication cost between the nodes [23, 20, 25]: each node can parallelise computation and/or store RDF data using our approach.

Materialisation has P-complete data complexity and is thus believed to be inherently sequential. Nevertheless, many practical parallelisation techniques have been developed; we discuss these using the following OWL 2 RL example axioms and their translation into datalog rules.

$$A \sqsubseteq B \qquad A(x, y) \rightarrow B(x, y) \qquad (\text{R1})$$

$$C \circ E \sqsubseteq D \qquad C(x, y) \wedge E(y, z) \rightarrow D(x, z) \qquad (\text{R2})$$

$$D \circ E \sqsubseteq E \qquad D(x, y) \wedge E(y, z) \rightarrow C(x, z) \qquad (\text{R3})$$

Interquery parallelism identifies rules that can be evaluated in parallel. For example, rules (R2) and (R3) must be evaluated jointly since C and D are mutually dependent, but rule (R1) is independent since B is independent from C and D . Such an approach does not guarantee a balanced workload distribution: for example, the evaluation of (R2) and (R3) might be more costly than of (R1); moreover, the number of independent components (two in our example) limits the degree of parallelism. *Intraquery parallelism* assigns distinct rule instantiations to threads by constraining variables in rules to domain subsets [7, 21, 9, 29, 27]. For example, with N threads and assuming that all objects are represented as integers, the i th thread can evaluate (R1)–(R3) with $(x \bmod N = i)$ added to the rules’ antecedents. Such a *static* partitioning does not guarantee an even workload distribution due to data skew. Systems such as WebPIE [23], Marvin [20], C/MPI [25]; DynamiTE [24], and [13] use variants of these approaches to support OWL 2 RL fragments such as RDFS or pD* [22].

In contrast, we handle general, recursive datalog rules using a parallel variant of the seminaïve algorithm [2]. Each thread extracts a fact from the database and matches it to the rules; for example, given a fact $E(a, b)$, a thread will match it to atom $E(y, z)$ in rule (R2) and evaluate subquery $C(x, b)$ to derive the rule’s consequences, and it will handle rule (R3) analogously. We thus obtain independent subqueries, each of which is evaluated on a distinct thread. The difference in subquery evaluation times does not matter because the number of queries is proportional to the number of tuples, and threads are fully loaded. We thus partition rule instantiations *dynamically* (i.e., as threads become free), unlike static partitioning which is predetermined and thus susceptible to skew.

To support this idea in practice, an RDF storage scheme is needed that (i) supports efficient evaluation of subqueries, and (ii) can be efficiently updated in parallel. To satisfy (i), indexes over RDF data are needed. Hexastore [26] and RDF-3X [19] provide six-fold sorted indexes that support merge joins and allow for a high degree of data compression. Such approaches may be efficient if data is static, but data changes continuously during materialisation so maintaining sorted indexes or re-compressing data can be costly and difficult to parallelise. Storage schemes based on columnar databases [15] with vertical partitioning [1] suffer from similar problems.

To satisfy both (i) and (ii), we use hash-based indexes that can efficiently match all RDF atoms (i.e., RDF triples in which some terms are replaced with variables) and thus support the index nested loops join. Hash table access can be easily parallelised, which allows us to support ‘mostly’ *lock-free* [14] updates: most of the time, at least one thread is guaranteed to make progress regardless of the remaining threads; however, threads do occasionally resort to localised locking. Lock-free data structures are resilient to adverse thread scheduling and thus often parallelise better than lock-based ones. Compared to the sort-merge [3] and hash join [4] algorithms, the index nested loops join with hash indexes exhibits random memory access which is potentially less efficient than sequential access, but our experiments suggest that hyperthreading and a high degree of parallelism can compensate for this drawback.

We have implemented our approach in a new system called *RDFox* and have evaluated its performance on several synthetic and real-world datasets. Parallelisation was beneficial in all cases, achieving a speedup in materialisation times of up to 13.9 with 16 physical cores, rising up to 19.3 with 32 virtual cores obtained by hyperthreading.

Our system also proved competitive with OWLIM-Lite (a commercial RDF system) and our implementation of the semi-naïve algorithm without parallelisation on top of PostgreSQL and MonetDB, with the latter systems running on a RAM disk. We did not independently evaluate query answering; however, queries are continuously answered during materialisation, so we believe that our results show that our data indexing scheme also supports efficient query answering over RDF data.

2 Preliminaries

A *term* is a *resource* (i.e., a constant) or a variable; we denote terms with t , and variables with x , y , and z . An (*RDF*) *atom* is a triple $\langle s, p, o \rangle$ of terms called the *subject*, *predicate*, and *object*, respectively. A *fact* is a variable-free atom. A *rule* r has the form (1), where H is the *head atom*, and B_1, \dots, B_n are *body atoms*; we let $h(r) := H$ and $b_i(r) := B_i$.

$$B_1 \wedge \dots \wedge B_n \rightarrow H \quad (1)$$

Rules must be *safe*: each variable in H must occur in some B_i . A *program* P is a finite set of possibly recursive rules. The *materialisation* (i.e., the fixpoint) $P^\infty(I)$ of a finite set of facts I with P , a *substitution* σ and its application $A\sigma$ to an atom A , and the composition $\tau\theta$ of substitutions τ and θ are defined as usual [2].

Our RDF system does not use the fixed OWL 2 RL/RDF rule set [18, Section 4.3]. Instead, we translate the OWL 2 RL axioms of the given KB into a datalog program; such programs generally contain simpler rules with fewer body atoms and are thus easier to evaluate. Our approach is, however, also applicable to OWL 2 RL/RDF rules.

3 Parallel Datalog Materialisation

We now present our algorithm that, given a datalog program P and a finite set of facts I , computes $P^\infty(I)$ on N threads. The data structure storing these facts (which we identify with I) must support several abstract operations: $I.add(F)$ should check whether I contains a fact F and add it to I if not; moreover, I should provide an iterator facts_I where $\text{facts}_I.next$ returns a not yet returned fact or ε if such a fact does not exist, and $\text{facts}_I.hasNext$ returns true if I contains a not yet returned fact. These operations need not enjoy the ACID¹ properties, but they must be *linearisable* [14]: 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. Accesses to I thus does not require external synchronisation via locks or critical sections.

Furthermore, I must support an interface for answering conjunctive queries constrained to a subset of I ; the latter will be used to prevent repeated derivations by a rule. To formalise this, we assume that I can be viewed as a vector; then, for $F \in I$ a fact, $I^{<F}$ contains all facts that come before F , and $I^{\leq F} := I^{<F} \cup \{F\}$. We make

¹ Atomicity, Consistency, Isolation, Durability; These four properties are assumed to hold true in database transactions and ensure their reliable execution.

no assumptions on the order in which facts_I returns the facts; the only requirement is that, once $\text{facts}_I.\text{next}$ returns a fact F , further additions should not change $I^{\leq F}$ —that is, returning F should ‘freeze’ $I^{\leq F}$. An *annotated query* is a conjunction of RDF atoms $Q = A_1^{\diamond_1} \wedge \dots \wedge A_k^{\diamond_k}$ where $\diamond_i \in \{<, \leq\}$. For F a fact and σ a substitution, $I.\text{evaluate}(Q, F, \sigma)$ returns the set containing each substitution τ such that $\sigma \subseteq \tau$ and $A_i\tau \in I^{\diamond_i F}$ for each $1 \leq i \leq k$. Such calls are valid only when set $I^{\leq F}$ is ‘frozen’ and hence does not change via additions.

Finally, for F a fact, $P.\text{rulesFor}(F)$ is the set containing all $\langle r, Q_i, \sigma \rangle$ where r is a rule in P of the form (1), σ is a substitution with $B_i\sigma = F$ for some $1 \leq i \leq n$, and

$$Q_i = B_1^< \wedge \dots \wedge B_{i-1}^< \wedge B_{i+1}^{\leq} \wedge \dots \wedge B_n^{\leq}. \quad (2)$$

To implement this operation efficiently, we index the rules of P using two hash tables H_1 and H_2 that map resources to sets. Each body atom B_i in a rule r obtained as in [10] is of the form (i) $\langle t, \text{rdf:type}, C \rangle$, in which case we add $\langle r, i, Q_i \rangle$ to $H_1[C]$, or (ii) $\langle t_1, R, t_2 \rangle$ with $R \neq \text{rdf:type}$, in which case we add $\langle r, i, Q_i \rangle$ to $H_2[R]$. Then, to compute $P.\text{rulesFor}(\langle s, p, o \rangle)$, we iterate over $H_1[o]$ if $p = \text{rdf:type}$ or $H_2[p]$ otherwise, and for each $\langle r, i, Q_i \rangle$ we determine whether a substitution σ exists such that $B_i\sigma = \langle s, p, o \rangle$. This strategy can be adapted to the OWL 2 RL/RDF rules.

To compute $P^\infty(I)$, we initialise a global counter W of waiting threads to 0 and let each of the N threads execute Algorithm 1. In lines 2–5, a thread acquires an unprocessed fact F (line 2), iterates through each rule r and each body atom B_i that can be mapped to F (line 3), evaluates the instantiated annotated query (line 4), and, for each query answer, instantiates the head of the rule and adds it to I (line 5). This can be seen as a fact-at-a-time version of the seminaïve algorithm [2]: set $\{F\}$ plays the role of the ‘delta-old’ relation; atoms $B_k^<$ in (2) are matched to ‘old’ facts (i.e., facts derived before F); and atoms B_k^{\leq} are matched to the ‘current’ facts (i.e., facts up to and including F). Like the seminaïve algorithm, our algorithm does not repeat derivations: each rule instantiation is considered at most once (but both algorithms can rederive the same fact using different rule instantiations).

A thread failing to extract a fact waits until either new facts are derived or all other threads reach the same state. This termination check is performed in a critical section (lines 7–15) implemented via a mutex m ; only idle threads enter the critical section, so the overhead of mutex acquisition is small. Counter W is incremented in line 7 and decremented in line 14 so, inside the critical section, W is equal to the number of threads processing lines 7–13. We increment W before acquiring m since, otherwise, termination will rely on fairness of mutex acquisition: the operating system may spuriously wake the threads waiting in line 13, forcing them, in this way, to continuously acquire the mutex m , and hence potentially preventing $W = N$ from becoming true. If $W = N$ holds in line 9, then all other threads are waiting in lines 7–13 and cannot produce more facts, so termination is indicated (line 10) and all waiting threads are woken up (line 11). Otherwise, a thread waits in line 13 for another thread to either produce a new fact or detect termination. The loop in lines 8–13 ensures that a thread stays inside the critical section and does not decrement W even if it is woken up but no work is available. Theorem 1 captures the correctness of our algorithm, and its proof is given in the online technical report <https://krr-nas.cs.ox.ac.uk/2014/DL/RDFox/paper.pdf>.

Algorithm 1 The Parallel Materialisation Algorithm

```

1: while  $run$  do
2:   while  $F := facts_I.next$  and  $F \neq \epsilon$  do
3:     for all  $\langle r, Q_i, \sigma \rangle \in P.rulesFor(F)$  do
4:       for all  $\tau \in I.evaluate(Q_i, F, \sigma)$  do
5:          $I.add(h(r)\tau)$ 
6:   increment  $W$  atomically
7:   acquire  $m$ 
8:   while  $facts_I.hasNext = false$  and  $run$  do
9:     if  $W = N$  then
10:       $run := false$ 
11:      notify all waiting threads
12:     else
13:       release  $m$ , await notification, acquire  $m$ 
14:   decrement  $W$  atomically
15:   release  $m$ 

```

Algorithm 2 $I.nestedLoops(Q, F, \tau, j)$

```

1: if  $j$  is larger than the number of atoms in  $Q$  then
2:   output  $\tau$ 
3: else
4:   let  $B_j^{\diamond j}$  be the  $j$ -th atom of  $Q$ 
5:   for all  $\theta$  such that  $B_j\tau\theta \in I^{\diamond j}$  do
6:      $I.nestedLoops(Q, F, j + 1, \tau \cup \theta)$ 

```

Theorem 1 *Algorithm 1 terminates and computes $P^\infty(I)$; moreover, each combination of r and τ is considered in line 5 at most once, so derivations are not repeated.*

Procedure $I.evaluate(Q_i, F, \sigma)$ in line 3 can use any join method, but our system uses a left to right evaluation of the body atoms (index nested loops). To this end, we reorder the atoms of each Q_i to obtain an efficient left-to-right join order Q'_i and then store Q'_i in our rule index; we use a simple greedy strategy, but any known planning algorithm can be used too. We then implement line 4 by calling $I.nestedLoops(Q'_i, F, \sigma, 1)$ defined in Algorithm 2. The latter critically depends on efficient matching of atoms in I , which we discuss in Section 4.

4 RAM-Based Storage of RDF Data

We next describe a main-memory RDF indexing scheme that (i) can efficiently match RDF atoms in line 5 of Algorithm 2, but also (ii) supports concurrent updates. The RDF stores presented in [26] and [19] satisfy (i) using ordered, compressed indexes, but maintaining the ordering can be costly when the data changes continuously and concurrently. Instead, we index the data using hash tables, which allows us to make insertions ‘mostly’ lock-free.

As is common in RDF systems, we encode resources as integers using a dictionary. We store encoded triples in a six-column *triple table* shown in Figure 1. Columns R_s , R_p , and R_o contain the integer encodings of the subject, predicate, and object of each triple. Each triple participates in three linked lists: an *sp*-list connects all triples with the same R_s grouped (but not necessarily sorted) by R_p , an *op*-list connects all triples with the same R_o grouped by R_p , and a *p*-list connects all triples with the same R_p without any grouping; columns N_{sp} , N_{op} , and N_p contain the next-pointers. Triple pointers are implemented as offsets into the triple table.

We next discuss RDF atom matching and the indexes used. Note that RDF atoms have eight different ‘binding patterns’. We can match $\langle x, y, z \rangle$ (only variables) by iterating over the triple table; if, for example, $x = y$, we skip triples with $R_s \neq R_p$. For the remaining seven patterns, we maintain six indexes of pointers into the *sp*-, *op*- and *p*-lists. Index I_{spo} contains each triple in the table, and so it can match RDF facts $\langle s, p, o \rangle$. Index I_s maps each s to the head $I_s[s]$ of the respective *sp*-list; to match an RDF atom

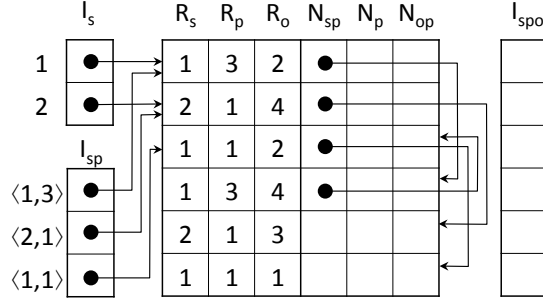


Fig. 1: Data Structure for Storing RDF Triples

$\langle s, y, z \rangle$ in I , we look up $I_s[s]$ and traverse the sp -list to its end; if $y = z$, we skip triples with $R_p \neq R_o$. Index I_{sp} maps each s and p to the first triple $I_{sp}[s, p]$ in an sp -list with $R_s = s$ and $R_p = p$; to match an RDF atom $\langle s, p, z \rangle$ in I , we look up $I_{sp}[s, p]$ and traverse the sp -list to its end or until we encounter a triple with $R_p \neq p$. We could match the remaining RDF atoms analogously using indexes I_p and I_{po} , and I_o and I_{os} ; however, in our experience, RDF atoms $\langle s, y, o \rangle$ occur rarely in queries, and I_{os} can be as big as I_{spo} since RDF datasets rarely contain more than one triple connecting the same s and o . Therefore, we use instead indexes I_o and I_{op} to match $\langle x, y, o \rangle$ and $\langle x, p, o \rangle$, and an index I_p to match $\langle x, p, z \rangle$. Finally, we match $\langle s, y, o \rangle$ by iterating over the sp - or op -list skipping over triples with $R_s \neq s$ or $R_o \neq o$; we keep in $I_s[s]$ and $I_o[o]$ the sizes of the two lists and choose the shorter one. To restrict any of these matches to $I^{<F}$ or $I^{\leq F}$, we compare the pointer to F with the pointer to each matched tuple, and we skip the matched tuple if necessary.

Indexes I_s , I_p , and I_o are realised as arrays. Indexes I_{sp} , I_{op} , and I_{spo} are realised as open addressing hash tables storing triple pointers, and they are doubled in size when the fraction of used buckets exceeds some factor f . To determine worst-case memory usage per triple (excluding the dictionary), let n be the number of triples; let p and r be the numbers of bytes used for pointers and resources; and let d_{sp} and d_{op} be the numbers of distinct sp - and op -groups divided by n . Each triple uses $3(r + p)$ bytes in the triple table. Index I_{spo} holds up to $\text{size}(I_{spo}) \cdot f = n$ many triples and uses most memory per triple just after resizing: $\text{size}(I_{spo}) = 2n/f$ buckets then require $2p/f$ bytes per triple. Analogously, worst-case per-triple memory usage for I_{sp} and I_{op} is $d_{sp} \cdot 2p/f$ and $d_{op} \cdot 2p/f$. Finally, I_s , I_p , and I_o are usually much smaller than n so we disregard them. Thus, for the common values of $r = 4$, $p = 8$, $f = 0.7$, $d_{sp} = 0.5$, and $d_{po} = 0.4$, we need at most 80 bytes per triple; this drops to 46 bytes for $p = 4$ (but then we can store at most 2^{32} triples).

4.1 ‘Mostly’ Lock-Free Insertion of Triples

Lock-freedom is usually achieved using compare-and-set: $\text{CAS}(loc, exp, new)$ loads the value stored at location loc into a temporary variable old , stores new into loc if $old = exp$, and returns old ; hardware ensures that these steps are atomic (i.e., without thread interference). Inserting triples lock-free is difficult as one must atomically query I_{spo} , add the triple to the table, and update I_{spo} . CAS does not directly support atomic modification of multiple locations, so descriptors [8] or multiword-CAS [12] are

Algorithm 3 add-triple(s, p, o)

Input:
 s, p, o : the components of the triple to be inserted

```

1:  $i := \text{hash}(s, p, o) \bmod |I_{spo}.buckets|$ 
2: do
3:   If needed, handle resize and recompute  $i$ 
4:   while  $T := I_{spo}.buckets[i]$  and  $T \neq \text{null}$  do
5:     if  $T = \text{INS}$  then continue
6:     if  $\langle T.R_s, T.R_p, T.R_o \rangle = \langle s, p, o \rangle$  then
7:       return
8:      $i := (i + 1) \bmod |I_{spo}.buckets|$ 
9:   while  $\text{CAS}(I_{spo}.buckets[i], \text{null}, \text{INS}) \neq \text{null}$ 
10:  Let  $T_{new}$  point to a fresh triple in the triple table
11:   $T_{new}.R_s := s, T_{new}.R_p := p, T_{new}.R_o := o$ 
12:   $I_{spo}.buckets[i] := T_{new}$ 
13:  Update all remaining indexes

```

Algorithm 4 insert-sp-list(T_{new}, T)

Input:
 T_{new} : pointer to the newly inserted triple
 T : pointer to the triple that T_{new} comes after

```

1: do
2:    $T_{next} := T.N_{sp}$ 
3:    $T_{new}.N_{sp} := T_{next}$ 
4:   while  $\text{CAS}(T.N_{sp}, T_{next}, T_{new}) \neq T_{next}$ 

```

needed. The latter techniques can be costly, so we instead resort to localised locking, cf. Algorithm 3. Here, $I_{spo}.buckets$ is the bucket array of the I_{spo} index; $|I_{spo}.buckets|$ is the array's length; and, for T a triple pointer, $T.R_s$ is the subject component of the triple that T points to, and $T.R_p, T.R_o, T.N_{sp}, T.N_p$, and $T.N_{op}$ are defined analogously.

Lines 1–12 of Algorithm 3 follow the standard approach for updating hash tables with open addressing: we determine the first bucket index (line 1), and we scan the buckets until we find an empty one (lines 4–8) or encounter the triple being inserted (line 7). The main difference to the standard approach is that, once we find an empty bucket, we need to lock it so that we can allocate a new triple. This is commonly done by introducing a separate lock that guards access to a range of buckets [14]. We, however, avoid the overhead of separate locks by storing into the bucket a special marker INS (line 9), and we make sure that other threads do not skip the bucket until the marker is removed (line 5). We lock the bucket using CAS so only one thread can claim it, and we reexamine the bucket if CAS fails as another thread could have just added the same triple. If CAS succeeds, we allocate a new triple T_{new} (line 10); if the triple table is big enough this requires only an atomic increment and is thus lock-free. Finally, we initialise the new triple (line 11), we store T_{new} into the bucket (line 12), and we update all remaining indexes (line 13).

To resize the bucket array (line 3), a thread locks the index, allocates a new array, raises a resize flag, and unlocks the index. Any thread that accesses the index first checks whether the resize flag is raised; if so, it keeps transferring blocks of 1024 buckets from the old array into the new array (which can be done lock-free since triples already exist in the table) until all buckets have been transferred; the thread processing the last block deallocates the old bucket array and resets the resize flag. Resizing is thus divided among threads and is lock-free, apart from the array allocation step.

To update the I_{sp} index in line 13, we scan its buckets as in Algorithm 3. If we find a bucket containing some T with $T.R_s = s$ and $T.R_p = p$, we insert T_{new} into the sp -list after T , which can be done lock-free as shown in Algorithm 4: we identify the triple T_{next} that follows T in the sp -list (line 2), we modify $T_{new}.N_{sp}$ so that T_{next} comes after T_{new} (line 3), and we update $T.N_{sp}$ to T_{new} (line 4); if another thread modifies $T.N_{sp}$ in the meantime, we repeat the process. If we find an empty bucket while scanning I_{sp} , we store T_{new} into the bucket and make T_{new} the head of $I_s[s]$; since this requires multiword-CAS, we again use local locks: we store INS into the

bucket of I_{sp} , we update I_s lock-free analogously to Algorithm 4, and we store T_{new} into the bucket of I_{sp} thus unlocking the bucket.

We update I_{op} and I_o analogously, and we update I_p lock-free as in Algorithm 4. Updates to all indexes are independent, which promotes concurrency.

4.2 Reducing Thread Interference

Each processor/core in modern systems has its own cache so, when core A writes to a memory location cached by core B , the cache of B is invalidated. If A and B keep writing into a shared location, cache synchronisation can significantly degrade the performance of parallel algorithms. Our data structure exhibits two such bottlenecks: each triple $\langle s, p, o \rangle$ is added at the end of the triple table; moreover, it is added after the first triple in the sp - and op -groups, which changes the next-pointer of the first triple.

To address the first bottleneck, each thread reserves a block of space in the triple table. When inserting $\langle s, p, o \rangle$, the thread writes $\langle s, p, o \rangle$ into a free location T_{new} in the reserved block, and it updates I_{spo} using a variant of Algorithm 3: since T_{new} is known *beforehand*, one can simply write T_{new} into $I_{spo}.buckets[i]$ in line 9 using CAS; moreover, if one detects in line 7 that I_{spo} already contains $\langle s, p, o \rangle$, one can simply reuse T_{new} later. Different threads thus write to distinct portions of the triple table, which reduces memory contention; moreover, allocating triple space in advance allows Algorithm 3 to become fully lock-free.

To address the second bottleneck, each thread i maintains a ‘private’ hash table I_{sp}^i holding ‘private’ insertion points for s and p . To insert triple $\langle s, p, o \rangle$ stored at location T_{new} , the thread determines $T := I_{sp}^i[s, p]$. If $T = null$, the thread inserts T_{new} into the global indexes I_{sp} and I_s as usual and sets $I_{sp}^i[s, p] := T_{new}$; thus, T_{new} becomes a ‘private’ insertion point for s and p in thread i . If $T \neq null$, the thread adds T_{new} after T ; since T is ‘private’ to thread i , updates are interference-free and do not require CAS. Furthermore, for each s the thread counts the triples $\langle s, p, o \rangle$ it derives, and it uses $I_{sp}^i[s, p]$ only once this count exceeds 100. ‘Private’ insertion points are thus maintained only for commonly occurring subjects, which keeps the size of I_{sp}^i manageable. Thread i analogously maintains a ‘private’ index I_{op}^i .

The former optimisation introduces a problem: when $facts_I$ eventually reaches a reserved block, the block cannot be skipped since further additions into the reserved space would invalidate the assumptions behind Theorem 1. So, when $facts_I$ reaches a reserved block, all empty rows in the reserved blocks are invalidated, and later ignored by $facts_I$, and from this point onwards all triples are added at the end of the table.

5 Evaluation

We implemented and evaluated a new system called **RDFox**; all datasets, systems, scripts, and test results are available online.² We evaluated RDFox in three ways: we investigated how materialisation scales with the number of threads; we compared a sequential version (i.e., without CAS) of RDFox with the concurrent version on a single thread to estimate the overhead of concurrency support; and we compared RDFox

² <http://www.cs.ox.ac.uk/isg/tools/RDFox/tests/>

Table 1: Test Datalog Programs and RDF Graphs

	LUBM			UOBM		DBpedia		Claros		
Ontology	\mathcal{O}_L	\mathcal{O}_{LE}	\mathcal{O}_U	\mathcal{O}_L	\mathcal{O}_U	\mathcal{O}_L	\mathcal{O}_{LE}	\mathcal{O}_L	\mathcal{O}_{LE}	\mathcal{O}_U
Rules	98	107	122	407	518	7,258	7,333	2,174	2,223	2,614
RDF Graph	01K	05K	010	01K						
Resources	32.9M	164.3M	0.4M	38.4M		18.7M			6.5M	
Triples	133.6M	691.1M	2.2M	254.8M		112.7M			18.8M	

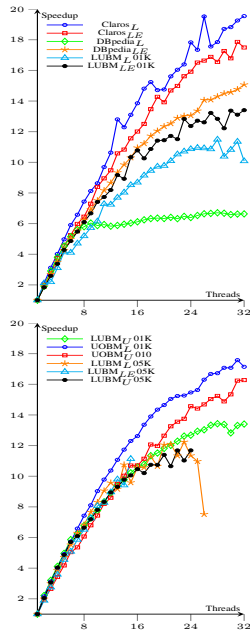
with other RDF systems. Unfortunately, no system we know of targets our setting exactly: many do not support materialisation of recursive rules [1, 26, 19, 31] and many are disk-based [6]. Hence, we compared RDFox with the following systems that, we believe, cover a wide range of approaches.

OWLIM-Lite, version 5.3, is a commercial RDF system developed by Ontotext. It stores triples in RAM, but keeps the dictionary on disk, and we configured it to evaluate our custom datalog programs, rather than the fixed OWL 2 RL/RDF rule set. Ontotext have confirmed that this is a fair way to use their system.

DBRDF is an RDF store we implemented on top of PostgreSQL (PG) 9.2 and MonetDB (MDB), Feb 2013-SP3 release. DBRDF can use either the vertical partitioning (VP) [1] or the triple table (TT) [5, 28] scheme for storing RDF data in a relational database. Neither PG nor MDB support recursive datalog rules, so DBRDF implements the seminaïve algorithm by translating datalog rules into DDL and SQL update statements and executing them sequentially; parallelising the latter is out of scope of this paper. More detail is given in <https://krr-nas.cs.ox.ac.uk/2014/DL/RDFox/paper.pdf>.

Table 1 summarises our test datasets. LUBM [11] and UOBM [17] are synthetic datasets so, for a parameter n , one can generate RDF graphs $LUBM_n$ and $UOBM_n$. The DBPedia dataset contains information about Wikipedia entities. Finally, Claros integrates cultural heritage data using a common vocabulary. The ontologies of LUBM, UOBM, and Claros are not in OWL 2 RL. In [30] such an ontology \mathcal{O} is converted into programs \mathcal{O}_L and \mathcal{O}_U such that $\mathcal{O}_U \models \mathcal{O}$ and $\mathcal{O} \models \mathcal{O}_L$; thus, \mathcal{O}_L (\mathcal{O}_U) captures a *lower* (*upper*) bound on the consequences of \mathcal{O} . The former is a natural test case since most RDF systems consider only \mathcal{O}_L , and the latter is interesting because of its complex rules. In order to make reasoning more challenging and to enlarge the materialised data set, we enriched \mathcal{O}_L to \mathcal{O}_{LE} with complex chain rules that encode relations specific to the domain of \mathcal{O} ; for example, we defined in $DBPedia_{LE}$ teammates as pairs of football players playing for the same team. We identify each test by combining the names of the datalog program and the RDF graph, such as $LUBM_{LE}$ 01K.

We tested RDFox on a Dell computer with 128 GB of RAM, 64-bit Red Hat Enterprise Linux Server 6.3 kernel version 2.6.32, and two Xeon E5-2650 processors with 16 physical cores, extended to 32 virtual cores via *hyperthreading* (i.e., when cores maintain separate state but share execution resources). In our comparison tests, we used a Dell computer with 128 GB of RAM, 64-bit CentOS 6.4 kernel version 2.6.32, and two Xeon E5-2643 processors with 8/16 cores. Each test involved importing an RDF graph and materialising a datalog program, and we recorded the wall-clock times for import and materialisation, the final number of triples, and the memory usage before and after materialisation. RDFox was allowed to use at most 100 GB of RAM, and it stored triple pointers using four bytes. To mitigate the overhead of disk access, we stored the



	Claros _L		Claros _{LE}		DBpedia _L		DBpedia _{LE}		LUBM _L 01K		LUBM _{LE} 01K	
	Time	Spd	Time	Spd	Time	Spd	Time	Spd	Time	Spd	Time	Spd
seq	1907	1.3	4128	1.2	158	1.0	8457	1.1	68	1.1	913	1.0
1	2477	1.0	4989	1.0	161	1.0	9075	1.0	73	1.0	947	1.0
2	1177	2.1	2543	2.0	84	1.9	4699	1.9	35	2.1	514	1.8
8	333	7.4	773	6.5	28	5.8	1453	6.2	14	5.2	155	6.1
16	179	13.9	415	12.0	26	6.1	828	11.0	8	8.7	88	10.8
24	139	17.8	313	15.9	25	6.4	695	13.1	7	10.9	77	12.4
32	127	19.5	285	17.5	24	6.6	602	15.1	7	10.1	71	13.4
Seq. imp.	61	1.1	57		331		335		421		408	
Par. imp.	58	-4.9%	58	1.7%	334	0.9%	367	9.5%	415	1.4%	415	1.7%
Triples	95.5 M		555.1 M		118.3 M		1529.7 M		182.4 M		332.6 M	
Memory	4.2 GB		18.0 GB		6.1 GB		51.9 GB		9.3 GB		13.8 GB	
Active	93.3%		98.8%		28.1%		94.5%		66.5%		90.3%	

	LUBM _L 01K		LUBM _L 05K		LUBM _{LE} 05K		LUBM _L 05K		UOBM _L 010		UOBM _L 01K	
	Time	Spd	Time	Spd	Time	Spd	Time	Spd	Time	Spd	Time	Spd
seq	122	1.1	422	1.0	4464	1.1	580	1.1	2381	1.2	476	1.1
1	135	1.0	442	1.0	4859	1.0	635	1.0	2738	1.0	532	1.0
2	61	2.2	221	2.0	2574	1.9	310	2.1	1400	2.0	247	2.2
8	20	6.8	65	6.8	745	6.5	96	6.6	451	6.1	72	7.4
16	13	10.6	42	10.6	Mem		60.7	10.5	256	10.7	42	12.6
24	11	12.7	39	11.3	Mem		54.3	11.7	188	14.6	34	15.5
32	10	13.4	Mem		Mem		Mem		168	16.3	31	17.1
Seq. imp.	410		2610		2587		2643		6		798	
Par. imp.	415	1.2%	2710	3.8%	3553	37%	2733	3.4%	6	0.0%	783	-1.9%
Triples	219.0 M		911.2 M		1661.0 M		1094.0 M		35.6 M		429.6 M	
Memory	11.1 GB		49.0 GB		75.5 GB		53.3 GB		1.1 GB		20.2 GB	
Active	85.3%		66.5%		90.3%		85.3%		99.7%		99.1%	

Fig. 2: Scalability and Concurrency Overhead of RDFox (All Times Are in Seconds)

Table 2: Comparison of RDFox with DBRDF and OWLIM-Lite (All Times Are in Seconds)

	RDFox		PG-VP		MDB-VP		PG-TT		MDB-TT		OWLIM-Lite	
	Import T B/t	Materialise T B/t	Import T B/t	Materialise T B/t	Import T B/t	Materialise T B/t	Import T B/t	Import T B/t	Import T B/t	Materialise T B/t	Import T B/t	Materialise T B/t
Claros _L	48	89	2062	60	1138	165	25026	97	883	58	Mem	Mem
Claros _{LE}			4218	44	Mem		Mem		1174	217	896	94
DBpedia _L			143	67	5844	148	Mem	Mem	5968	174	4492	92
DBpedia _{LE}	274	69	9538	49	Mem		Mem		1735	171	Mem	Mem
LUBM _L 01K			71	65	948	127	27	41				
LUBM _{LE} 01K	332	74	765	54	15632	112	6136	56	7947	187	5606	104
LUBM _L 01K			113	67	Time		138	44			2409	40
UOBM _L 010	5	68	2501	43	Time		96	50	120	203	96	85
UOBM _L 01K	632	64	467	63	13864	119	6708	107	11063	41	358	33
									14901	176	10892	101
									4778	38	Time	Time

databases of MDB and PG on a 100 GB RAM disk. For OWLIM-Lite, we kept the dictionary on a 50 GB RAM disk; since the system materialises rules during import, we loaded each RDF graph once with the test program and once with no program and we subtracted the two times; Ontotext confirmed that this yields a good materialisation time estimate. Each test was limited to 10 hours, and we report averages over three runs.

Figure 2 shows the speedup of RDFox with the number of used threads. The middle part of the tables shows the sequential and parallel import times, and the percentage slowdown for the parallel version. The lower part of the tables shows the number of triples and memory consumption after materialisation (the number of triples before is given in Table 1), and the percentage of *active* triples (i.e., triples to which a rule was applied). Import times differ by at most 5% between the sequential and parallel version. For materialisation, the overhead of lock-free updates is between 10% and 30%, so parallelisation pays off already with two threads. With all 16 physical cores, RDFox achieves a speedup of up to 13.9; this increases to 19.5 with 32 virtual cores, suggesting

that hyperthreading and a high degree of parallelism can mitigate the effect of CPU stalls due to random memory access. The flattening of the speedup curves is due to the limited capabilities of virtual cores, and the fact that each thread contributes to system bus congestion. Per-thread indexes (see Section 4) proved very effective at reducing thread interference, although they did cause memory exhaustion in some tests; however, the comparable performance of the sequential version of RDFox (which does not use such indexes) suggests that the cost of maintaining them is not high. The remaining source of interference is in the calls to `factsI.next`, which are more likely to overlap with many threads and few active triples. The correlation between the speedup for 32 threads and the percentage of active triples is 0.9, explaining the low speedup on `DBpediaL`. The excessive parallel import time observed for `LUBMLE05K` is due to a glitch and should be in the same range as `LUBML05K` and `LUBMU05K`. Since every thread keeps its private insertion points (cf. Section 4.2), the memory intensive `LUBMs05K` test cases run out of memory for higher numbers of threads.

Table 2 compares RDFox with OWLIM-Lite and DBRDF on PG and MDB with the VP or TT scheme. Columns T show the times in seconds, and columns B/t show the number of bytes per triple. Import in DBRDF is about 20 times slower than in RDFox, but half of this time is used by the Jena RDF parser. VP is 33% more memory efficient than TT, as it does not store triples’ predicates, and MDB-VP can be up to 34% more memory-efficient than RDFox; however, MDB-TT is not, which is surprising since RDFox does not compress data. We do not know how OWLIM-Lite splits the dictionary between the disk and RAM, so the RAM consumption in Table 2 is a ‘best-case’ estimate. On materialisation tests, both MDB-TT and PG-TT ran out of time in all but one case (MDB-TT completed `DBpediaL` in 11,958 seconds): as observed in [1], self-joins on the triple table are notoriously difficult for RDBMSs. In contrast, although it implements TT, RDFox successfully completed all tests. MDB-VP was faster than RDFox on two tests (`LUBML01K` and `UOBML01K`); however, it was slower on the others, and it ran out of memory on many tests. PG-VP was always much slower, and it could not complete many tests.

6 Conclusion & Outlook

We presented a novel and very efficient approach to parallel materialisation of datalog in centralised, multi-core, main-memory RDF systems. However, when equality is axiomatised and thus explicated in the materialised data, equality cliques cause a quadratic blowup. We would like to address this challenge by using native equality reasoning in which reasoning is performed over a factorised representation of the data. We further intend to combine native equality reasoning with incremental reasoning techniques. Our goals also include adapting the RDF indexing scheme to secondary storage, the main difficulty of which will be to reduce random access.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB Journal* 18(2), 385–406 (2009)
2. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*. Addison Wesley (1995)
3. Albutiu, M.C., Kemper, A., Neumann, T.: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB* 5(10), 1064–1075 (2012)
4. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: *ICDE*. pp. 362–373 (2013)
5. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: *ISWC*. pp. 54–68 (2002)
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: *VLDB*. pp. 1216–1227 (2005)
7. Dong, G.: On Distributed Processability of Datalog Queries by Decomposing Databases. *SIGMOD Record* 18(2), 26–35 (1989)
8. Fraser, K., Harris, T.L.: Concurrent Programming Without Locks. *ACM TOCS* 25(2), 1–61 (2007)
9. Ganguly, S., Silberschatz, A., Tsur, S.: A Framework for the Parallel Processing of Datalog Queries. In: *SIGMOD*. pp. 143–152 (1990)
10. Grosz, B.N., Horrocks, I., Volz, R., Decker, S.: Description Logic Programs: Combining Logic Programs with Description Logic. In: *WWW*. pp. 48–57 (2003)
11. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. *JWS* 3(2-3), 158–182 (2005)
12. Harris, T.L., Fraser, K., Pratt, I.A.: A Practical Multi-word Compare-and-Swap Operation. In: *DISC*. pp. 265–279 (2002)
13. Heino, N., Pan, J.Z.: RDFS Reasoning on Massively Parallel Hardware. In: *ISWC*. pp. 133–148 (2012)
14. Herlihy, M., Shavit, N.: *The Art of Multiprocessor Programming*. Morgan Kaufmann (2008)
15. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K., Kersten, M.: MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35(1), 40–45 (2012)
16. Larson, P.A.: Letter from the Special Issue Editor. *IEEE Data Engineering Bulletin, Special Issue on Main-Memory Database Systems* 36(2), 5 (2013)
17. Ma, L., Yang, Y., Qiu, Z., Xie, G.T., Pan, Y., Liu, S.: Towards a Complete OWL Ontology Benchmark. In: *ESWC*. pp. 125–139 (2006)
18. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: *OWL 2 Web Ontology Language: Profiles*, W3C Recommendation (2009)
19. Neumann, T., Weikum, G.: The RDF-3X Engine for Scalable Management of RDF Data. *VLDB Journal* 19(1), 91–113 (2010)
20. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed Reasoning over Large-scale Semantic Web Data. *JWS* 7(4), 305–316 (2009)
21. Seib, J., Lausen, G.: Parallelizing Datalog Programs by Generalized Pivoting. In: *PODS*. pp. 241–251 (1991)
22. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *JWS* 3(2–3), 79–115 (2005)
23. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *JWS* 10, 59–75 (2012)
24. Urbani, J., Margara, A., Jacobs, C.J.H., van Harmelen, F., Bal, H.E.: Dynamite: Parallel materialization of dynamic rdf data. In: *ISWC*. pp. 657–672 (2013)

25. Weaver, J., Hendler, J.A.: Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In: ISWC. pp. 682–697 (2009)
26. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. PVLDB 1(1), 1008–1019 (2008)
27. Wolfson, O., Ozeri, A.: Parallel and Distributed Processing of Rules by Data-Reduction. IEEE TKDE 5(3), 523–530 (1993)
28. 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: ICDE. pp. 1239–1248 (2008)
29. Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE TKDE 7(1), 163–176 (1995)
30. 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: WWW. pp. 1569–1580 (2013)
31. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: Answering SPARQL Queries via Subgraph Matching. PVLDB 4(8), 482–493 (2011)