Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems

Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks and Dan Olteanu
Department of Computer Science, Oxford University
Oxford, United Kingdom
firstname.lastname@cs.ox.ac.uk

Abstract

We present a novel approach to parallel materialisation (i.e., fixpoint computation) of datalog programs in centralised, main-memory, multi-core RDF systems. Our approach comprises an 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: with 16 physical cores, materialisation can be up to 13.9 times faster than with just one core.

1 Introduction

Querying OWL 2 RL ontologies can be reduced to answering datalog queries. The latter problem can be solved by backward chaining (Abiteboul, Hull, and Vianu 1995; Urbani et al. 2012), or one can materialise all consequences of the rules and the data so that queries can be answered without the rules. Since materialisation supports efficient querying, it is commonly used in practice, but it is also 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 (Larson 2013) 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 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 (Urbani et al. 2012; Oren et al. 2009): each node can parallelise computation and/or store RDF data using our approach.

Materialisation is PTIME-complete in data complexity and is thus believed to be inherently sequential. Nevertheless, many practical parallelisation techniques have been developed, and we discuss them using the following rules.

\[
\begin{align*}
A(x, y) &\rightarrow B(x, y) \quad (R1) \\
C(x, y) \land E(y, z) &\rightarrow D(x, z) \quad (R2) \\
D(x, y) \land E(y, z) &\rightarrow C(x, z) \quad (R3)
\end{align*}
\]

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 (Dong 1989; Seib and Lausen 1991; Ganguly, Silberschatz, and Tsir 1990; Zhang, Wang, and Chau 1995; Wolfson and Ozeri 1993). For example, given N threads and assuming that objects are encoded as integers, the i\textsuperscript{th} thread can evaluate \(R1\)–\(R3\) with \((x \mod N = i)\) added to the rules’ antecedents. Such a static partitioning does not guarantee an even workload distribution due to data skew. Shao, Bell, and Hull (1991) exploit both inter- and intraquery parallelism in datalog, and Gupta et al. (1986) do so in the context of production systems. The latter approaches, however, are unlikely to be applicable to large data sets since Rete networks—data structures used to manage rules in production systems—store partial rule instantiations and thus often use large amounts of memory.

In the RDF setting, systems such as WebPIE (Urbani et al. 2012), Marvin (Oren et al. 2009), C/MPI (Weaver and Hendler 2009), DynamiTE (Urbani et al. 2013), by Heino and Pan (2012), and by Goodman and Mizell (2010) use variants of the above mentioned approaches. All of these handle only OWL 2 RL fragments such as RDFS or pD\(^*\) (ter Horst 2005), but can generally handle RDFS without (much) communication between the threads. Furthermore, Soma and Prasanna (2008) exploit both inter- and intraquery parallelism for distributed OWL 2 RL reasoning.

In contrast, we handle general, recursive datalog rules using a parallel variant of the seminaive algorithm (Abiteboul, Hull, and Vianu 1995). 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 \(C(x, a)\) to derive atoms of the form \(D(x, b)\), 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
large (i.e., proportional to the number of tuples) so threads are fully loaded. We thus partition rule instantiations dynamically (i.e., as threads become free), unlike static partitioning which is determined in advance 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 (Weiss, Karras, and Bernstein 2008) and RDF-3X (Neumann and Weikum 2010) 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 (Iliadis et al. 2012) with vertical partitioning (Abadi et al. 2009) 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 joins. Hash table access can be easily parallelised, which allows us to support ‘mostly’ lock-free (Herlihy and Shavit 2008) 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 (Albutiu, Kemper, and Neumann 2012) and hash join (Balkesen et al. 2013) 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 supports efficient query answering over RDF data. Each rule must be safe: each variable in $H$ must occur in some $B_i$. A program $P$ is a finite set of possibly recursive rules. A substitution $\sigma$, its application $A\sigma$ to an atom $A$, and the composition $\tau\theta$ of substitutions $\tau$ and $\theta$ are all defined as usual (Abiteboul, Hull, and Vianu 1995).

Let $I$ be a finite set of facts. Given a rule $r$, $r(I)$ is the smallest set containing $h(r)\sigma$ for each substitution $\sigma$ such that $b_i(r)\sigma \in I$ for each $1 \leq i \leq \text{len}(r)$. Given a datalog program $P$, let $P(I) := \bigcup_{r \in P} r(I)$; let $P^0(I) := I$; for each $i > 0$, let $P^i(I) := P^{i-1}(I) \cup P^i(I)$; and let $P^\infty(I) := \bigcup P^i(I)$. The latter set is the materialisation of $I$ with $P$, and its computation is the topic of this paper.

OWL 2 RL is a fragment of OWL 2 for which reasoning can be implemented using datalog techniques. Two styles of OWL 2 RL reasoning are commonly used. 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 (Motik et al. 2009, 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 (Grosof et al. 2003), but whose rules are shorter and simpler. This approach is complete only if the data does not contain ‘nonsensical’ triples such as $(\text{rdf:type}, \text{rdf:type}, \text{rdf:type})$—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.

# 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.

## Intuition

A global counter $W$ of waiting threads is first initialised to zero. Then, each of the $N$ threads extracts an unprocessed fact $F$ and considers each rule $r \in P$ of the form (1). For each $B_i$ in the body of $r$ unifiable with $F$ using a substitution $\sigma$, the thread computes the partially instantiated rule

$$B_1\sigma \land \ldots \land B_{i-1}\sigma \land B_{i+1}\sigma \land \ldots \land B_n\sigma \rightarrow H\sigma$$

(2)

and then matches the body of (2) to the available facts; for each match, it adds the instantiated rule head to the facts (unless already present). The thread repeats these steps until it cannot extract an unprocessed fact, upon which it increments $W$ and goes to sleep. When another thread produces a fact, the sleeping thread wakes up, decrements $W$, and continues matching rules. All facts have been processed when $W$ reaches $N$, and then all threads terminate.

A naive application of this idea would be inefficient: with rule $(x, \text{rdf:type}, A) \land (x, \text{rdf:type}, B) \rightarrow (x, \text{rdf:type}, C)$ and facts $(a, \text{rdf:type}, A)$ and $(a, \text{rdf:type}, B)$, we would derive $(a, \text{rdf:type}, C)$ twice—that is, we would consider the same rule instance twice. As a remedy, we match the body atoms in (2) only to the facts that appear before the extracted fact $F$, which ensures that derivations are not repeated.

Matching the body of (2) can be solved using an arbitrary join algorithm. Merge join is preferred in RDF.
databases (Weiss, Karras, and Bernstein 2008; Neumann and Weikum 2010), but it requires data to be sorted; since the data changes continuously during materialisation, maintaining the ordering can be costly. In Section 4 we show how to index the data using hash-based indexes that allow for efficient parallel updates, but that can also match each instantiated body atom in \(O(1)\) time. Then, we match \(2\) using index nested loop join with ‘sideways information passing’.

Facts are assigned to threads as they are extracted, so our algorithm essentially exploits intraquery parallelism without any need for load balancing. We see only two situations in which our approach would not distribute the workload to threads equitably. First, applying a single rule to a single fact \(F\) may dominate the total computation time so that most of the computation is done by one thread, but this is extremely unlikely in practice. Second, the rules may be inherently serial, as in the following example.

\[
P = \{ \langle x, \text{rdf:type}, A \rangle \land \langle x, R, y \rangle \rightarrow \langle y, \text{rdf:type}, A \rangle \} \quad (3)
\]

\[
I = \{ \langle a_0, \text{rdf:type}, A \rangle, \langle a_0, R, a_1 \rangle, \ldots, \langle a_{-1}, R, a_0 \rangle \} \quad (4)
\]

For each \(1 \leq i \leq \ell\), fact \(\langle a_{i-1}, \text{rdf:type}, A \rangle\) is needed to derive \(\langle a_i, \text{rdf:type}, A \rangle\); hence, our algorithm, as well as all approaches known to us, become sequential. The existence of such cases is unsurprising since datalog materialisation is \(PTIME\)-complete and is believed to be inherently serial.

**Formalisation**

Our algorithm manipulates the set of facts \(I\) using operations that we next describe from an abstract point of view; in Section 4, we then discuss how to support these operations efficiently. If \(I\) does not contain a fact \(F\), then \(I.add(F)\) adds \(F\) to \(I\) and returns true; otherwise, \(I.add(F)\) returns false. Moreover, \(I\) must support iteration over its facts: \(I.next\) returns a previously not returned fact from \(I\), or \(\varepsilon\) if such a fact does not exist; and \(I.hasNext\) returns true if \(I\) contains a previously not returned fact. These operations need not enjoy the ACID properties, but they must be *linearisable* (Herlihy and Shavit 2008): 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 do not require external synchronisation via locks or critical sections.

Moreover, \(I\) 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, \(I^{\leq F}\) contains all facts that come before \(F\), and \(I^{\leq F} := I^{< F} \cup \{ F \}\). We make no assumptions on the order in which \(I.next\) returns the facts; the only requirement is that, once it 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

\[
Q = A_1^{\tau_1} \land \ldots \land A_k^{\tau_k}\ 
\text{where } \forall i \in \{1, \ldots, k\}.\ 
\]

For \(F\) a fact and \(\sigma\) a substitution, \(I.evaluate(Q, F, \sigma)\) returns the set containing each substitution \(\tau\) such that \(\sigma \subseteq \tau\) and \(A_i^{\tau_i} \in F^{< F}\) 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, \(P.rulesFor(F)\) is the set containing all \(\langle r, Q_i, \sigma \rangle\) where \(r\) is a rule in \(P\) of the form (1), \(\sigma\) is a substitution such that \(B_i^{\sigma} = F\) for some \(1 \leq i \leq n\), and

\[
Q_i = B_1^{\leq} \land \ldots \land B_{i-1}^{\leq} \land B_i^{\geq} \land \ldots \land B_n^{\geq}.\ 
\] (5)

Iterating over \(P\) to identify the rules matching \(F\) would be very inefficient; however, RDF atoms in rules often contain constants, which we can exploit to retrieve the candidate rules more efficiently. In particular, we index the rules of \(P\) using a hash table \(H\) mapping facts to sets. In particular, for each rule \(r \in P\) and each body atom \(B_i\) in \(r\), let \(B_i'\) be obtained by replacing all variables in \(B_i\) with a special symbol \(*\); then, we add \(\langle r, i, Q_i \rangle\) to \(H[B_i']\). To compute \(P.rulesFor(F)\), we generate all eight triples obtained by replacing some resources in \(F\) with \(*\); then, for each thus obtained triple \(F'\) and each \(\langle r, i, Q_i \rangle \in H[F']\), we determine whether a substitution \(\sigma\) exists such that \(B_i^{\sigma} = F\).

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 2–6, 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 semi-naive algorithm (Abiteboul, Hull, and Vianu 1995): set \(\{F\}\) plays the role of the ‘delta-old’ relation; atoms \(B_i^{\leq}\) in (5) are matched to the ‘old’ facts (i.e., facts derived before \(F\)); and atoms \(B_i^{\geq}\) are matched to the ‘current’ facts (i.e., facts up to and including \(F\)). Like the semi-naive 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 must wait until either new facts are derived or all other threads reach the same state. This termination check is performed in a critical section (lines 8–16) 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 15 so, inside the critical section, \(W\) is equal to the number of threads processing lines 8–14; since \(W\) is incremented before acquiring \(m\), termination does not depend on fairness of mutex acquisition. If \(W = N\) holds in line 10, then all other threads are waiting in lines 8–14 and cannot produce more facts, so termination is indicated (line 11) and all waiting threads are woken up (line 12). Otherwise, a thread waits in line 14 for another thread to either produce a new fact or detect termination. The loop in lines 9–14 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.

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

**Proof.** Our discussion of the termination condition (lines 7–16) in Section 3 shows that all threads terminate only when \(I.hasNext\) returns false. Moreover, the number of different facts that our algorithm can derive is finite since it is determined by the number of resources in \(I\) and \(P\), and duplicates
Algorithm 1 Threads of the Materialisation Algorithm

| Global: | I: a set of facts to materialise |
|         | P: a datalog program |
|         | N: the total number of threads |
|         | W: the number of waiting threads (initially 0) |
|         | m: a mutex variable |

1: while run do
2:   while F := I.next and F \neq \epsilon do
3:     for each \langle r, Q_i, \sigma \rangle \in P.rulesFor(F) do
4:         for each \tau \in I.evaluate(Q_i, F, \sigma) do
5:             if I.add(h(\tau)) and W > 0 then
6:                 notify all waiting threads
7:             increment W atomically
8:             acquire m
9:         while I.hasNext = false and run do
10:            if W = N then
11:                run := false
12:            else
13:                release m, await notification, acquire m
14:            decrement W atomically
15:        release m

are eliminated eagerly. Finally, I.next and I.add(F) are linearisable, so let \mathcal{F} = \{F_1, \ldots, F_m\} be the sequence of distinct facts extracted in line 2; by a slight abuse of notation, we treat \mathcal{F} as a set when the order of the elements in \mathcal{F} is irrelevant. We next show that \mathcal{F} = P^{\infty}(I).

One can show that \mathcal{F}_i \in P^{\infty}(I) holds for each 1 \leq i \leq m by a straightforward induction on i: either \mathcal{F}_i \in I, or \mathcal{F}_i is obtained by applying a rule r \in P to I^{\leq F_i}.

We next prove that F' \in \mathcal{F} holds for each i and each F' \in P^{\infty}(I). This is obvious for i = 0, so assume that the claim holds for some i and consider an arbitrary fact F' \in P^{i+1}(I) \setminus P^i(I). Then, a rule r \in P exists that produces F' by matching b_j(r) to some F_j \in P^i(I) for each 1 \leq j \leq len(r); from the induction assumption, we have F_j \in \mathcal{F} for each 1 \leq j \leq len(r). Now let F be the fact from \{F_1 \cup \ldots \cup F_m\} with the largest index in I, and let k be the smallest integer such that \mathcal{F}_k = F. Fact F is returned at some point in line 2, so the algorithm considers in line 3 the rule r, annotated query Q_k, and substitution \sigma such that b_k(r) = F. But then, I.evaluate(Q_k, F, \sigma) returns a substitution \tau such that b_k(r) \tau \in I^{< F} for each 1 \leq j \leq len(r), and so F' = h(\tau)\tau is derived in line 5, as required.

Finally, we prove by contradiction that derivations are not repeated. To this end, assume that F and F' are (not necessarily distinct) facts extracted in line 2, and let Q_i and Q'_i be annotated queries for the same rule r and substitution \tau considered in line 5. By the definition of Q_i and Q'_i, we have b_i(r) \tau = F and b_i(r) \tau = F'. We consider two cases.

Assume F = F', and w.l.o.g. assume that i \leq j. If i = j, we have a contradiction since F is extracted in line 2 only once and Q_i = Q'_j is considered in line 3 only once. If i < j, we have a contradiction since b_i = \epsilon holds in annotated query Q'_j, so atom b_i(r) cannot be matched to F in Q'_j (i.e., we cannot have b_i(r)\tau = F) due to F \notin I^{< F'}.

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}^{\tau} be the j-th atom of Q
5:   for each \theta such that B_{j}^{\tau} \theta \in P^{\infty}(F) do
6:     I.nestedLoops(Q, F, \tau \cup \theta, j + 1)

Assume F \neq F', and w.l.o.g. assume that F' occurs after F in I; thus, F' \notin I^{< F}. But then, b_j(r) \tau = F' leads to a contradiction since atom b_j(r) cannot be matched to F' in Q_j, when fact F is extracted in line 2. □

Procedure I.evaluate(Q_i, F, \sigma) in line 3 can use any join method, but our system uses 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, \tau, 1) shown in Algorithm 2. The latter critically depends on the efficient matching of atoms in I^{< F} and I^{\leq F}, 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. Weiss, Karras, and Bernstein (2008) and Neumann and Weikum (2010) 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 integer encodings of the subject, predicate, and object of each triple. We store the 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. There are eight different ‘binding patterns’ of RDF.
atoms. We can match pattern \((x, y, z)\) 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 to pointers into the \(sp\)-, \(op\)-, and \(p\)-lists. Index \(I spo\) contains each triple in the table, and so it can match RDF atoms \((s, p, o)\). Index \(I s\) maps each \(s\) to the head \(I s[s]\) of the respective \(sp\)-list; to match an RDF atom \((s, y, z)\) in \(I s\), 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 \(I p\) maps each \(s\) and \(p\) to the first triple \(I p[s, p]\) in an \(sp\)-list with \(R_s = s\) and \(R_p = p\); to match an RDF atom \((s, p, z)\) in \(I p\), we look up \(I p[s, p]\) 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 \(I spo\), and \(I o\) and \(I os\); however, in our experience, RDF atoms \((s, y, o)\) 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 \(I s\) and \(I op\) to match \((x, y, o)\) and \((x, p, o)\), and an index \(I p\) to match \((x, p, z)\). Finally, we match \((s, y, o)\) 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 \(T^P\) or \(T^F\), 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 indexed by resource IDs. 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 \(f\). Hash codes are obtained by combining the relevant resource IDs via Jenkins hashing. To determine the 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 \(I spo\) uses most memory per triple just after resizing: \(2n/f\) buckets then require \(2p/f\) bytes per triple. An analogous worst-case 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 \(d_{op} = 0.4\), 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).

‘Mostly’ Lock-Free Insertion of Triples

Lock-freedom is usually achieved using compare-and-set: \(CAS(loc, exp, new)\) loads the value stored at location \(loc\) into a temporary variable \(old\) stores the value of \(new\) into \(loc\) if \(old = exp\), and returns \(old\); hardware ensures that all steps are atomic (i.e., without interference). Lock-free triple insertion 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 (Fraser and Harris 2007) or multiword-CAS (Harris, Fraser, and Pratt 2002) are needed. These 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

\[
\begin{align*}
\text{Algorithm 3 add-triple}(s, p, o) & \\
\text{Input: } s, p, o: \text{the components of the inserted triple} & \\
1: & i := hash(s, p, o) \mod |I spo\text{.buckets}| \\
2: & \text{do} & \\
3: & \text{if needed, handle resize and recompute } i & \\
4: & \text{while } T := I spo\text{.buckets}[i] \text{ and } T \neq null \text{ do} & \\
5: & \text{if } T = \text{INS then continue} & \\
6: & \text{if } \langle T.R_s, T.R_p, T.R_o \rangle = \langle s, p, o \rangle \text{ then return} & \\
7: & i := (i + 1) \mod |I spo\text{.buckets}| & \\
8: & \text{while } CAS(I spo\text{.buckets}[i], null, \text{INS}) \neq null & \\
9: & \text{Let } T_{new} \text{ point to a fresh triple in the triple table} & \\
10: & T_{new}.R_s := s, T_{new}.R_p := p, T_{new}.R_o := o & \\
11: & I spo\text{.buckets}[i] := T_{new} & \\
12: & \text{Update all remaining indexes} & \\
\end{align*}
\]

pointer, \(T.R_s\) is the subject of the triple that \(T\) points to, and \(T.R_p, T.R_o, T.N_x, T.N_y, T.N_o\), and \(T.N_s\) are analogous.

Lines 1–11 of Algorithm 3 are similar to the standard approach for open hashing: we determine the first bucket index (line 1), and we scan the buckets until we find an empty one (line 4–7) or encounter the triple being inserted (line 6). The main difference is that, once we find an empty bucket, we 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 (Herlihy and Shavit 2008). We, however, avoid the overhead of separate locks by storing into the bucket a special marker \(\text{INS}\) (line 8), 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 9); 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 10), we store \(T_{new}\) into the bucket (line 11), and we update all remaining indexes (line 12).

To resize the bucket array (line 3), a thread temporarily locks the index, allocates a new array, initialises a global counter of 1024-bucket blocks in the bucket array, and unlocks the index. When any thread accesses the index, as long as the global counter is not zero, the thread transfers a block of 1024 buckets from the old array into the new array (which can be done lock-free since triples already exist in the table) and decrements the counter; moreover, the thread detecting that the counter dropped to zero deallocates the old bucket array. Resizing is thus divided among threads and is lock-free, apart from the array allocation step.

To update the \(I sp\) index in line 12, 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 \(T_{new}.N_x\) so that \(T_{next}\) comes after \(T_{new}\) (line 3), and we update \(T.N_x\) to \(T_{new}\) (line 4); if another thread modifies \(T.N_x\) 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 lo-
5 Evaluation

We implemented and evaluated a new system called RDFox. The system is written in C++, and it works on x86-64 computers running Windows, Linux, or Mac OS X. It provides three versions of our RDF storage scheme. The ‘sequential’ version does not use CAS operations; it represents triple pointers using 6 bytes, and so it can store at most $281 \cdot 10^{12}$ triples. The ‘parallel narrow’ and ‘parallel wide’ versions support concurrent access. To ensure atomicity of memory operations, they represent triple pointers using native word lengths of 4 and 8 bytes, respectively, and can thus store at most $4 \cdot 10^9$ and $18 \cdot 10^{18}$ triples, respectively. We used the ‘sequential’ and ‘parallel narrow’ versions in our tests.

We evaluated different aspects of our approach in the following four ways. First, we compared the ‘sequential’ and the ‘parallel’ versions of RDFox on a single thread in order to estimate the overhead of concurrency support. Second, we investigated how materialisation in the ‘parallel’ version of RDFox scales with the number of threads. Third, we compared RDFox with OWLIM-Lite—a commercial RDF system that also stores triples in RAM. Fourth, we investigated how our approach compares with state of the art approaches for RDF storage and materialisation based on relational databases. All datasets, test systems, scripts, and test results are available online.\(^1\)

Test Data

Table 1 summarises our test datasets. LUBM (Guo, Pan, and Heflin 2005) and UOBM (Ma et al. 2006) are synthetic datasets; given a parameter $n$, one can generate RDF graphs LUBM$n$ and UOBM$n$. DBpedia 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. Zhou et al. (2013) convert such an ontology $O$ into programs $O_L$ and $O_U$ such that $O_U \models O$ and $O \models O_L$; thus, $O_L$ ($O_U$) captures a lower (upper) bound on the consequences of $O$. Program $O_L$ is largely equivalent to the one obtained from $O$ using the transformation by Grossof et al. (2003), and is a natural test case since most RDF systems consider only $O_L$. Program $O_U$ is interesting because its rules are more complex. To obtain even more complex programs, we manually extended $O_L$ to $O_{LE}$ with chain rules that encode relations specific to the domain of $O$; for example, we defined in DBpedia$_{LE}$ ‘teammates’ as pairs of football players playing for the same team. Programs obtained from ontologies other than LUBM contain rules with the owl:sameAs property in the head, which we axiomatised explicitly—that is, we make it symmetric and transitive, and, for each property $R$ and class $C$ in $P$, we introduce the following rules.

\[\langle x, rdf:type, C\rangle \land \langle x, owl:sameAs, y\rangle \rightarrow \langle y, rdf:type, C\rangle\]
\[\langle x_1, R, x_2\rangle \land \langle x_1, owl:sameAs, y_1\rangle \rightarrow \langle y_1, R, x_2\rangle\]
\[\langle x_1, R, x_2\rangle \land \langle x_2, owl:sameAs, y_2\rangle \rightarrow \langle x_1, R, y_2\rangle\]

We identify each test by combining the names of the datalog program and the RDF graph, such as LUBM$_{LE}$ 01K.

\(^1\)http://www.cs.ox.ac.uk/igis/tools/RDFox/tests/
Comparison Systems

Most existing systems do not target our setting exactly; many cannot materialise recursive rules (Abadi et al. 2009; Weiss, Karras, and Bernstein 2008; Neumann and Weikum 2010; Zou et al. 2011) and many are disk-based (Chong et al. 2005). To our knowledge, OWLIM-Lite is the only system that supports recursive rules and stores triples in RAM, so we used it for a direct performance comparison. Furthermore, RDF is often managed using relational databases, so we implemented a new test system, DBRDF, that stores RDF using two well-known storage schemes and materialises recursive rules using the seminaïve algorithm. OWLIM-Lite and DBRDF do not parallelise computation, so we conducted all comparison tests on a single thread. Comparing RDFox with (possible) parallel variants of the known approaches might be interesting, but it is not strictly necessary: our comparison provides us with a baseline, and we show independently that RDFox parallelises computation very well. We next discuss the comparison systems in more detail.

OWLIM-Lite, version 5.3, is a commercial RDF system developed by Ontotext.² It stores triples in RAM, but keeps the dictionary on disk, so we stored the latter on a RAM disk drive. We configured OWLIM-Lite to use our custom datalog programs, rather than the fixed OWL 2 RL/RDF rule set. Due to simpler rules, OWLIM-Lite consistently processed the custom datalog programs several times faster than the fixed OWL 2 RL/RDF rules, so we do not believe that using custom programs put OWLIM-Lite at a disadvantage. Moreover, OWLIM-Lite always materialises rules during import so, to estimate the materialisation overhead, we loaded each RDF graph once with the test program and once with no program and subtracted the two times. Ontotext have confirmed that this is a fair way to use their system.

DBRDF simulates RDF systems based on row and column stores (Abadi et al. 2009; Broekstra, Kampman, and van Harmelen 2002; Wu et al. 2008) by storing RDF data in PostgreSQL (PG) 9.2 and MonetDB (MDB), Feb 2013-SP3 release. We stored the data on a RAM disk drive; while this clearly does not mimic the performance of a purely RAM-based system, it mitigates the overhead of disk access. On PG, we created all tables as UNLOGGED to eliminate the fault recovery overhead, and we configured the database to aggressively use memory using the following options:

fsync = off, synchronous_commit = off, full_page_writes = off, bgwriter_lru_pages = 0, shared_buffers = 16GB, work_mem = 16GB, effective_cache_size = 16GB.

DBRDF can store RDF data using one of the two well-known storage schemes. In the vertical partitioning (VP) variant (Abadi et al. 2009), a triple \( \langle s, \text{rdf:type}, C \rangle \) is stored as tuple \( \langle s \rangle \) in a unary relation \( C \), and a triple \( \langle s, R, o \rangle \) with \( R \neq \text{rdf:type} \) is stored as tuple \( \langle s, o \rangle \) in a binary relation \( R \).

In the triple table (TT) variant (Chong et al. 2005; Broekstra, Kampman, and van Harmelen 2002), each triple \( \langle s, p, o \rangle \) is stored directly in a common ternary table, and the latter is indexed on \( \langle s, p, o \rangle \), \( \langle p, o \rangle \), and \( \langle o, s \rangle \). In either case, the data was not clustered explicitly: MDB manages data disposition internally, and clustering in PG makes no sense as data changes continuously during materialisation.

Neither PG nor MDB support recursive datalog rules, so DBRDF implements the seminaïve algorithm. Given a datalog program \( P \), DBRDF analyses the dependency graph (Abiteboul, Hull, and Vianu 1995) of \( P \) and partitions the latter into programs \( P_1, \ldots, P_n \) such that \( P_\infty(\ldots P_\infty(\ldots P_\infty(I)\ldots)) \); then, DBRDF applies the seminaive algorithm (Abiteboul, Hull, and Vianu 1995) to each \( P_1 \). The ‘old’, ‘delta-old’, and ‘new’ relations from the seminaive algorithm are implemented as temporary tables. The rules in \( P_1 \) are translated into INSERT INTO \( \ldots \) SELECT \( \ldots \) statements; for example, with VP, rule \( \langle x, R, y \rangle \land \langle y, R, z \rangle \rightarrow \langle x, S, z \rangle \) is translated as follows:

```
INSERT INTO S(s,o) SELECT DISTINCT t1.s,t2.o FROM R t1, R t2 WHERE t1.o = t2.s AND NOT EXISTS (SELECT 1 FROM S s WHERE s = t1.s AND t2.o = t2.o)
```

Here, DISTINCT AND NOT EXISTS eliminate duplicates, which is essential for termination and not repeating derivations. All statements were evaluated using the ‘read uncommitted’ transaction isolation level. On PG, SQL statements were combined into a PL/pgSQL script that is evaluated sequentially inside the database; all scripts used in our evaluation are available online. On MBD, SQL statements were issued sequentially from a Java program. MBD can parallelise query evaluation, but we observed that this achieved a speedup of at most two; thus, for fairness we restricted the number of threads in all tests to one.

Test Setting

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). This computer provided us with many cores for the speedup tests, but we could not run MBD and PG on it for administrative reasons; hence, we ran the comparison tests on another 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. RDFox was allowed to use all 100 GB of RAM. We stored the databases of MBD and PG on a 100 GB RAM disk and allowed the rest to be used as the systems’ working memory; thus, PG and MBD could use more RAM in total than RDFox. We kept the dictionaries of OWLIM-Lite on a 50 GB RAM disk. Each test involved importing an RDF graph and materialising a datalog program, and we recorded the wall-clock times for import and

---

²http://www.ontotext.com/
materialisation, the final number of triples, and the memory usage before and after materialisation. Each run was limited to 10 hours, and we report averages over three runs.

**Test Results**

The graphs and Table (a) in Figure 2 show the speedup of RDFox with the number of used threads. The middle part of Table (a) shows the sequential and parallel import times, and the percentage slowdown for the parallel version. The lower part of Table (a) 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). As one can see, import times differ by at most 5\% between the sequential and the parallel version. For materialisation, the overhead of lock-free updates is between 10\% and 30\%, so parallelisation pays off even with just two threads. With all 16 physical cores, RDFox is up to 13.9 times faster than with just one core; 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 more limited capabilities of virtual cores, and the fact that each thread contributes to system bus congestion. Per-thread interference (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 \( f \) next, which are more likely to overlap when there are many threads but 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.

Table (b) in Figure 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. We do not know how OWLIM-Lite splits the dictionary between the disk and RAM, so its memory consumption is a 'best-case' estimate. Import in DBRDF is about 20 times slower than in RDFox, but half of import 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-PP is not as efficient as RDFox, which is surprising since RDFox does not compress data. 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): self-joins on the triple table are notoriously difficult for RDBMSs (Abadi et al. 2009). MDB-VP was always much slower, and it also could not complete many tests. In contrast, RDFox successfully completed all tests, although it essentially implements TT.

![Figure 2: Results of our Empirical Evaluation (All Times Are in Seconds)](image-url)
Acknowledgements
This work was supported by the EPSRC projects ExODA and MaSI3, and the Oxford Supercomputing Centre (OSC).

References