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 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 (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.

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

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

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

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 (Dong 1989; Seib and Lausen 1991; Ganguly, Silberschatz, and Tsur 1990; Zhang, Wang, and Chau 1995; Wolfson and Ozeri 1993). For example, given N threads and assuming that objects are encoded as integers, the ith 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. 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 seminaïve 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 (Idreos 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 loops join. 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 proved competitive with OWLIM-Lite (a commercial RDF system) and our implementation of the seminaïve 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. Unless otherwise stated, s, p, o, and t are terms, and x, y, and z are variables. 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 $B_1, \ldots, B_n$ are *body atoms*; let $\mathsf{len}(r) := n$, $\mathsf{h}(r) := H$, and $\mathsf{b}_i(r) := B_i$ for each $1 \leq i \leq n$.

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

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* σ, its application $A\sigma$ to an atom A, and the composition $\tau\theta$ of substitutions τ and θ 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 $\mathsf{h}(r)\sigma$ for each substitution σ such that $\mathsf{b}_i(r)\sigma \in I$ for each $1 \leq i \leq \mathsf{len}(r)$. Given a datalog program P, let $P(I) := \bigcup_{r \in P} r(I)$; let $P^0(I) := I$; for each $i > 0$, let $P^i(I) := P^{i-1}(I) \cup P(P^{i-1}(I))$; and let $P^\infty(I) := \bigcup_i P^i(I)$. 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 $\langle$*rdf:type*, *rdf:type*, *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.

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 σ, the thread computes the partially instantiated rule

$$B_1\sigma \wedge \ldots \wedge B_{i-1}\sigma \wedge B_{i+1}\sigma \wedge \ldots \wedge B_n\sigma \rightarrow H\sigma \tag{2}$$

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 naïve application of this idea would be inefficient: with rule $\langle x, \textit{rdf:type}, A\rangle \wedge \langle x, \textit{rdf:type}, B\rangle \rightarrow \langle x, \textit{rdf:type}, C\rangle$ and facts $\langle a, \textit{rdf:type}, A\rangle$ and $\langle a, \textit{rdf:type}, B\rangle$, we would derive $\langle a, \textit{rdf:type}, C\rangle$ 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.

$$P = \{\langle x, \textit{rdf:type}, A\rangle \wedge \langle x, R, y\rangle \rightarrow \langle y, \textit{rdf:type}, A\rangle\} \quad (3)$$

$$I = \{\langle a_0, \textit{rdf:type}, A\rangle, \langle a_0, R, a_1\rangle, \ldots, \langle a_{\ell-1}, R, a_\ell\rangle\} \quad (4)$$

For each $1 \leq i \leq \ell$, fact $\langle a_{i-1}, \textit{rdf:type}, A\rangle$ is needed to derive $\langle a_i, \textit{rdf:type}, A\rangle$; 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 $I.\mathsf{add}(F)$ adds F to I and returns `true`; otherwise, $I.\mathsf{add}(F)$ returns `false`. Moreover, I must support iteration over its facts: $I.\mathsf{next}$ returns a previously not returned fact from I, or ϵ if such a fact does not exist; and $I.\mathsf{hasNext}$ 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 does 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, $I^{<F}$ contains all facts that come before F, and $I^{\leq F} := I^{<F} \cup \{F\}$. We make no assumptions on the order in which $I.\mathsf{next}$ 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 $Q = A_1^{\bowtie_1} \wedge \ldots \wedge A_k^{\bowtie_k}$ where $\bowtie_i \in \{<, \leq\}$. For F a fact and σ a substitution, $I.\mathsf{evaluate}(Q, F, \sigma)$ returns the set containing each substitution τ such that $\sigma \subseteq \tau$ and $A_i\tau \in I^{\bowtie_i F}$ 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, $P.\mathsf{rulesFor}(F)$ 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 such that $B_i\sigma = F$ for some $1 \leq i \leq n$, and

$$Q_i = B_1^{<} \wedge \cdots \wedge B_{i-1}^{<} \wedge B_{i+1}^{\leq} \wedge \ldots \wedge B_n^{\leq}. \quad (5)$$

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 $P.\mathsf{rulesFor}(F)$, 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 σ 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 seminaïve algorithm (Abiteboul, Hull, and Vianu 1995): set $\{F\}$ plays the role of the 'delta-old' relation; atoms $B_k^{<}$ in (5) are matched to the '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 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 τ 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 $I.\mathsf{hasNext}$ 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)
run: a Boolean flag (initially true)
m: a mutex variable

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

are eliminated eagerly. Finally, $I.\mathsf{next}$ and $I.\mathsf{add}(F)$ are linearisable, so let $\mathcal{F} = \langle F_1, \ldots, F_m \rangle$ 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 $F_i \in P^\infty(I)$ holds for each $1 \leq i \leq m$ by a straightforward induction on i: either $F_i \in I$, or F_i is obtained by applying a rule $r \in P$ to $I^{<F_i}$.

We next prove that $F' \in \mathcal{F}$ holds for each i and each $F' \in P^i(I)$. 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 $\mathsf{b}_j(r)$ to some $F^j \in P^i(I)$ for each $1 \leq j \leq \mathsf{len}(r)$; from the induction assumption, we have $F^j \in \mathcal{F}$ for each $1 \leq j \leq \mathsf{len}(r)$. Now let F be the fact from $\{F^j | 1 \leq j \leq \mathsf{len}(r)\}$ with the largest index in I, and let k be the smallest integer such that $F^k = F$. 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 σ such that $\mathsf{b}_k(r)\sigma = F$. But then, $I.\mathsf{evaluate}(Q_k, F, \sigma)$ returns a substitution τ such that $\mathsf{b}_j(r)\tau \in I^{\leq F}$ for each $1 \leq j \leq \mathsf{len}(r)$, and so $F' = \mathsf{h}(r)\tau$ 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'_j be annotated queries for the same rule r and substitution τ considered in line 5. By the definition of Q_i and Q'_j, we have $\mathsf{b}_i(r)\tau = F$ and $\mathsf{b}_j(r)\tau = F'$. 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 $\bowtie_i = <$ holds in annotated query Q'_j, so atom $\mathsf{b}_i(r)$ cannot be matched to F in Q'_j (i.e., we cannot have $\mathsf{b}_i(r)\tau = F$) due to $F \notin I^{<F'}$.

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

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

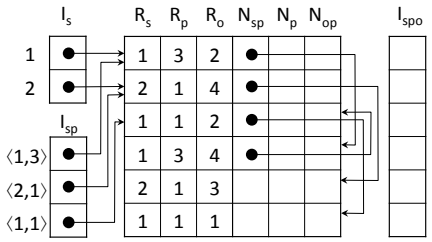


Figure 1: Data Structure for Storing RDF Triples

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

Procedure $I.\mathsf{evaluate}(Q_i, F, \sigma)$ 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 $I.\mathsf{nestedLoops}(Q'_i, F, \sigma, 1)$ 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 IDs using a dictionary. IDs are produced by a counting sequence, and so they can be used as array indexes.

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 $\langle x, y, z\rangle$ 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 $\langle s, p, o\rangle$. Index I_s maps each s to the head $I_s[s]$ of the respective sp-list; to match an RDF atom $\langle s, y, z\rangle$ 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 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 Jenkings 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. Analogously, 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 $d_{po} = 0.4$, 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).

'Mostly' Lock-Free Insertion of Triples

Lock-freedom is usually achieved using compare-and-set: $\mathsf{CAS}(loc, exp, new)$ 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, $I_{spo}.\mathsf{buckets}$ is the bucket array of the I_{spo} index; $|I_{spo}.\mathsf{buckets}|$ is the array's length; and, for T a triple

Algorithm 3 $\mathsf{add{-}triple}(s, p, o)$

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

```
1:  i := hash(s, p, o) mod |I_spo.buckets|
2:  do
3:      If needed, handle resize and recompute i
4:      while T := I_spo.buckets[i] and T ≠ null do
5:          if T = INS then continue
6:          if ⟨T.R_s, T.R_p, T.R_o⟩ = ⟨s, p, o⟩ then return
7:          i := (i + 1) mod |I_spo.buckets|
8:  while CAS(I_spo.buckets[i], null, INS) ≠ null
9:  Let T_new 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.buckets[i] := T_new
12: Update all remaining indexes
```

pointer, $T.R_s$ is the subject 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 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 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 $T_{new}.N_{sp}$ 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 lo-

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** CAS$(T.N_{sp}, T_{next}, T_{new}) \neq T_{next}$

cal 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 I_{spo}, I_p, I_{sp} and I_s, and I_{op} and I_o are independent, which promotes concurrency.

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 always added after the first triple in the *sp*- and *op*-groups, which changes the next-pointer of the first triple.

For 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 8 using CAS; moreover, if one detects in line 6 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.

For the second bottleneck, each thread i maintains a 'private' hash table I^i_{sp} holding 'private' insertion points into the *sp*-lists. To insert triple $\langle s, p, o\rangle$ stored at location T_{new}, the thread determines $T := I^i_{sp}[s, p]$. If $T =$ *null*, the thread inserts T_{new} into the global indexes I_{sp} and I_s as usual and sets $I^i_{sp}[s, p] := T_{new}$; 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 $I^i_{sp}[s, p]$ only once this count exceeds 100. 'Private' insertion points are thus maintained only for commonly occurring subjects, which keeps the size of I^i_{sp} manageable. Thread i analogously maintains a 'private' index I^i_{op}.

The former optimisation introduces a problem: when I.next eventually reaches a reserved block, the block cannot be skipped since further additions into the reserved space would invalidate the assumptions behind Theorem 1. Therefore, when I.next reaches a reserved block, all empty rows in the reserved blocks are marked as unused, and from this point onwards all triples are added at the end of the table; furthermore, I.next skips over all unused rows in the table.

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 LUBMn and UOBMn. 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 $\mathcal{O}$ 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 $\mathcal{O}$. Program $\mathcal{O}_L$ is largely equivalent to the one obtained from $\mathcal{O}$ using the transformation by Grosof et al. (2003), and is a natural test case since most RDF systems consider only $\mathcal{O}_L$. Program $\mathcal{O}_U$ is interesting because its rules are more complex. To obtain even more complex programs, we manually extended $\mathcal{O}_L$ to $\mathcal{O}_{LE}$ with 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. 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.

$$\langle x, \textit{rdf:type}, C\rangle \wedge \langle x, \textit{owl:sameAs}, y\rangle \rightarrow \langle y, \textit{rdf:type}, C\rangle$$
$$\langle x_1, R, x_2\rangle \wedge \langle x_1, \textit{owl:sameAs}, y_1\rangle \rightarrow \langle y_1, R, x_2\rangle$$
$$\langle x_1, R, x_2\rangle \wedge \langle x_2, \textit{owl:sameAs}, y_2\rangle \rightarrow \langle x_1, R, y_2\rangle$$

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/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		

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 $\langle s, \textit{rdf:type}, C\rangle$ is stored as tuple $\langle s\rangle$ in a unary relation C, and a triple $\langle s, R, o\rangle$ with $R \neq \textit{rdf:type}$ is stored as tuple $\langle s, o\rangle$ in a binary relation R. Moreover, each unary relation $C(s)$ is indexed on s, and each binary relation $R(s, o)$ is indexed on $\langle s, o\rangle$ and o. 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 $P^\infty(I) = P_n^\infty(\ldots P_1^\infty(I)\ldots)$; then, DBRDF applies the seminaïve algorithm (Abiteboul, Hull, and Vianu 1995) to each P_i. The 'old', 'delta-old', and 'new' relations from the seminaïve algorithm are implemented as temporary tables. The rules in P_i are translated into INSERT INTO ... SELECT ... statements; for example, with VP, rule $\langle x, R, y\rangle \wedge \langle y, R, z\rangle \rightarrow \langle x, S, z\rangle$ is translated as follows:

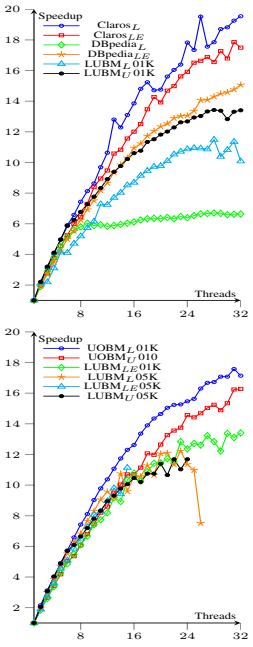
```
INSERT INTO S(s,o) SELECT DISTINCT t1.s, t2.o
FROM R t1, R t2 WHERE t1.o = t2.s AND NOT EXISTS
(SELECT 1 FROM S ex WHERE ex.s = t1.s AND ex.o = t2.o)
```

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 MDB, SQL statements were issued sequentially from a Java program. MDB 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 MDB 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 at most 100 GB of RAM. We stored the databases of MDB and PG on a 100 GB RAM disk and allowed the rest to be used as the systems' working memory; thus, PG and MDB could use more RAM in total than RDFox. We kept the dictionary 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

[2] http://www.ontotext.com/



(a) Scalability and Concurrency Overhead of RDFox

	Claros$_L$		Claros$_{LE}$		DBpedia$_L$		DBpedia$_{LE}$		LUBM$_L$ 01K		LUBM$_{LE}$ 01K		LUBM$_U$ 01K		LUBM$_L$ 05K		LUBM$_{LE}$ 05K		LUBM$_U$ 05K		UOBM$_U$ 010		UOBM$_L$ 01K	
	Time	Spd	Time	Spd	Time	Spd	Time	Spd	Time	Spd	Time	Spd	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	122	1.1	422	1.0	4464	1.1	580	1.1	2381	1.2	476	1.1
1	2477	1.0	4989	1.0	161	1.0	9075	1.0	73	1.0	947	1.0	135	1.0	442	1.0	4859	1.0	635	1.0	2738	1.0	532	1.0
2	1177	2.1	2543	2.0	84	1.9	4699	1.9	35	2.1	514	1.8	61	2.2	221	2.0	2574	1.9	310	2.1	1400	2.0	247	2.2
8	333	7.4	773	6.5	28	5.8	1453	6.2	14	5.2	155	6.1	20	6.8	65	6.8	745	6.5	96	6.6	451	6.1	72	7.4
16	179	13.9	415	12.0	26	6.1	828	11.0	8	8.7	88	10.8	13	10.6	42	10.6	Mem		60.7	10.5	256	10.7	42	12.6
24	139	17.8	313	15.9	25	6.4	695	13.1	7	10.9	77	12.4	11	12.7	39	11.3	Mem		54.3	11.7	188	14.6	34	15.5
32	127	19.5	285	17.5	24	6.6	602	15.1	7	10.1	71	13.4	10	13.4	Mem		Mem		Mem		168	16.3	31	17.1
Seq. imp.	61		57		331		335		421		408		410		2610		2587		2643		6		798	
Par. imp.	58	-4.9%	58	1.7%	334	0.9%	367	9.5%	415	1.4%	415	1.7%	415	1.2%	2710	3.8%	3553	37%	2733	3.4%	6	0.0%	783	-1.9%
Triples	95.5 M		555.1 M		118.3 M		1529.7 M		182.4 M		332.6 M		219.0 M		911.2 M		1661.0 M		1094.0 M		35.6 M		429.6 M	
Memory	4.2 GB		18.0 GB		6.1 GB		51.9 GB		9.3 GB		13.8 GB		11.1 GB		49.0 GB		75.5 GB		53.3 GB		1.1 GB		20.2 GB	
Active	93.3%		98.8%		28.1%		94.5%		66.5%		90.3%		85.3%		66.5%		90.3%		85.3%		99.7%		99.1%	

(b) Comparison of RDFox with DBRDF and OWLIM-Lite

	RDFox				PG-VP				MDB-VP				PG-TT		MDB-TT		OWLIM-Lite			
	Import		Materialise		Import		Materialise		Import		Materialise		Import		Import		Import		Materialise	
	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t
Claros$_L$	48	89	2062	60	1138	165	25026	97	883	58	Mem		1174	217	896	94	300	54	14293	28
Claros$_{LE}$			4218	44			Mem				Mem								Time	
DBpedia$_L$	274	69	143	67	5844	148	Mem		15499	51	354	49	5968	174	4492	92	1735	171	14855	164
DBpedia$_{LE}$			9538	49			Mem				Mem								Time	
LUBM$_L$01K			71	65			948	127			27	41							316	36
LUBM$_{LE}$ 01K	332	74	765	54	7736	144	15632	112	6136	56	Mem		7947	187	5606	104	2409	40	Time	
LUBM$_U$01K			113	67			Time				138	44							Time	
UOBM$_U$010	5	68	2501	43	116	154	Time		96	50	Mem		120	203	96	85	32	69	11311	24
UOBM$_L$01K	632	64	467	63	13864	119	6708	107	11063	41	358	33	14901	176	10892	101	4778	38	Time	

Figure 2: Results of our Empirical Evaluation (All Times Are in Seconds)

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 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 I.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 DBpedia$_L$.

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-TT 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 DBpedia$_L$ in 11,958 seconds): self-joins on the triple table are notoriously difficult for RDBMSs (Abadi et al. 2009). MDB-VP was faster than RDFox on two tests (LUBM$_L$01K and UOBM$_L$01K), but it often ran out of memory. PG-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.

6 Conclusion & Outlook

We presented an efficient novel approach to parallel materialisation of datalog rules in main-memory RDF databases. We see two main challenges for our future work. First, we shall extend the approach to handle the *owl:sameAs* property via rewriting—that is, by replacing all equal resources with a single representative. Second, we shall adapt the RDF indexing scheme to secondary storage. The main difficulty is to reduce the need for random access (e.g., to locate the keys of hash table entries or follow list pointers).

Acknowledgements

This work was supported by the EPSRC projects ExODA and MaSI^3, and the Oxford Supercomputing Centre (OSC).

References

Abadi, D. J.; Marcus, A.; Madden, S.; and Hollenbach, K. 2009. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB Journal* 18(2):385–406.

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison Wesley.

Albutiu, M.-C.; Kemper, A.; and Neumann, T. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB* 5(10):1064–1075.

Balkesen, C.; Teubner, J.; Alonso, G.; and Özsu, M. T. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 362–373.

Broekstra, J.; Kampman, A.; and van Harmelen, F. 2002. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, 54–68.

Chong, E. I.; Das, S.; Eadon, G.; and Srinivasan, J. 2005. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, 1216–1227.

Dong, G. 1989. On Distributed Processibility of Datalog Queries by Decomposing Databases. *SIGMOD Record* 18(2):26–35.

Fraser, K., and Harris, T. L. 2007. Concurrent Programming Without Locks. *ACM TOCS* 25(2):1–61.

Ganguly, S.; Silberschatz, A.; and Tsur, S. 1990. A Framework for the Parallel Processing of Datalog Queries. In *SIGMOD*, 143–152.

Goodman, E., and Mizell, D. 2010. Scalable In-memory RDFS Closure on Billions of Triples. In *Proc. SSWS*, volume 669 of *CEUR WS Proceedings*.

Grosof, B. N.; Horrocks, I.; Volz, R.; and Decker, S. 2003. Description Logic Programs: Combining Logic Programs with Description Logic. In *WWW*, 48–57.

Guo, Y.; Pan, Z.; and Heflin, J. 2005. Lubm: A benchmark for owl knowledge base systems. *JWS* 3(2-3):158–182.

Gupta, A.; Forgy, C.; Newell, A.; and Wedig, R. G. 1986. Parallel Algorithms and Architectures for Rule-Based Systems. In *Proc. ISCA*, 28–37.

Harris, T. L.; Fraser, K.; and Pratt, I. A. 2002. A Practical Multi-word Compare-and-Swap Operation. In *DISC*, 265–279.

Heino, N., and Pan, J. Z. 2012. RDFS Reasoning on Massively Parallel Hardware. In *ISWC*, 133–148.

Herlihy, M., and Shavit, N. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Idreos, S.; Groffen, F.; Nes, N.; Manegold, S.; Mullender, K.; and Kersten, M. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35(1):40–45.

Larson, P.-A. 2013. Letter from the Special Issue Editor. *IEEE Data Engineering Bulletin, Special Issue on Main-Memory Database Systems* 36(2):5.

Ma, L.; Yang, Y.; Qiu, Z.; Xie, G. T.; Pan, Y.; and Liu, S. 2006. Towards a Complete OWL Ontology Benchmark. In *ESWC*, 125–139.

Motik, B.; Cuenca Grau, B.; Horrocks, I.; Wu, Z.; Fokoue, A.; and Lutz, C. 2009. OWL 2 Web Ontology Language: Profiles, W3C Recommendation.

Neumann, T., and Weikum, G. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB Journal* 19(1):91–113.

Oren, E.; Kotoulas, S.; Anadiotis, G.; Siebes, R.; ten Teije, A.; and van Harmelen, F. 2009. Marvin: Distributed Reasoning over Large-scale Semantic Web Data. *JWS* 7(4):305–316.

Seib, J., and Lausen, G. 1991. Parallelizing Datalog Programs by Generalized Pivoting. In *PODS*, 241–251.

Shao, J.; Bell, D. A.; and Hull, M. E. C. 1991. Combining Rule Decomposition and Data Partitioning in Parallel Datalog Program Processing. In *Proc. PDIS*, 106–115.

Soma, R., and Prasanna, V. K. 2008. Parallel Inferencing for OWL Knowledge Bases. In *Proc. ICPP*, 75–82.

ter Horst, H. J. 2005. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *JWS* 3(2–3):79–115.

Urbani, J.; Kotoulas, S.; Maassen, J.; van Harmelen, F.; and Bal, H. 2012. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *JWS* 10:59–75.

Urbani, J.; Margara, A.; Jacobs, C. J. H.; van Harmelen, F.; and Bal, H. E. 2013. Dynamite: Parallel materialization of dynamic rdf data. In *ISWC*, 657–672.

Weaver, J., and Hendler, J. A. 2009. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *ISWC*, 682–697.

Weiss, C.; Karras, P.; and Bernstein, A. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB* 1(1):1008–1019.

Wolfson, O., and Ozeri, A. 1993. Parallel and Distributed Processing of Rules by Data-Reduction. *IEEE TKDE* 5(3):523–530.

Wu, Z.; Eadon, G.; Das, S.; Chong, E. I.; Kolovski, V.; Annamalai, M.; and Srinivasan, J. 2008. Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In *ICDE*, 1239–1248.

Zhang, W.; Wang, K.; and Chau, S.-C. 1995. Data Partition and Parallel Evaluation of Datalog Programs. *IEEE TKDE* 7(1):163–176.

Zhou, Y.; Cuenca Grau, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2013. Making the Most of Your Triple Store: Query Answering in OWL 2 Using an RL Reasoner. In *WWW*, 1569–1580.

Zou, L.; Mo, J.; Chen, L.; Özsu, M. T.; and Zhao, D. 2011. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB* 4(8):482–493.