RDFox: A Highly-Scalable RDF Store

Yavor Nenov[1](✉), Robert Piro[1], Boris Motik[1], Ian Horrocks[1], Zhe Wu[2], and Jay Banerjee[2]

[1] University of Oxford, Oxford, UK
{yavor.nenov,robert.piro,boris.motik,ian.horrocks}@cs.ox.ac.uk
[2] Oracle Corporation, Redwood Shores, CA, USA
{alan.wu,jayanta.banerjee}@oracle.com

Abstract. We present RDFox—a main-memory, scalable, centralised RDF store that supports materialisation-based parallel datalog reasoning and SPARQL query answering. RDFox uses novel and highly-efficient parallel reasoning algorithms for the computation and incremental update of datalog materialisations with efficient handling of owl:sameAs. In this system description paper, we present an overview of the system architecture and highlight the main ideas behind our indexing data structures and our novel reasoning algorithms. In addition, we evaluate RDFox on a high-end SPARC T5-8 server with 128 physical cores and 4TB of RAM. Our results show that RDFox can effectively exploit such a machine, achieving speedups of up to 87 times, storage of up to 9.2 billion triples, memory usage as low as 36.9 bytes per triple, importation rates of up to 1 million triples per second, and reasoning rates of up to 6.1 million triples per second.

1 Introduction

An increasing number of Semantic Web applications represent knowledge and data using the Resource Description Framework (RDF) [12]. Such applications use RDF stores to efficiently store large amounts of RDF data, manage background knowledge about a domain, and answer queries. The background knowledge is usually captured using an OWL 2 ontology [18], possibly extended with SWRL rules [10]. An ontology describes dependencies between entities, which allows an RDF store to enrich query answers with results not explicitly stated in the data. Queries are typically expressed in SPARQL [21], and the main computational service of RDF stores is to evaluate queries over both the explicit facts and the facts implied by the background knowledge. Answering queries with respect to arbitrary OWL 2 ontologies is often infeasible in practice due to the high computational complexity of the logical formalisms that underpin OWL 2 [9]. OWL 2 profiles [13] deal with intractability by restricting the expressivity of the ontology language in a way that enables efficient query answering over large datasets. OWL 2 RL is one such profile that is supported, at least to an extent, by many state of the art RDF stores. Consequences of OWL 2 RL ontologies can be captured using *datalog* [1]—a rule-based language developed by both the database and the knowledge representation communities. Queries over a datalog program and a dataset can be answered in several

M. Arenas et al. (Eds.): ISWC 2015, Part II, LNCS 9367, pp. 3–20, 2015.
DOI: 10.1007/978-3-319-25010-6_1

different ways. In scenarios where the performance of query answering is critical, a common approach is to precompute and explicitly store all consequences of the program and the dataset so that subsequent queries can be evaluated without any further reference to the program. This approach is also known as *materialisation*, and it is used in state of the art systems such as GraphDB [3] and Oracle's RDF store [22].

In this system description paper we present RDFox—a highly scalable, centralised, main-memory RDF store that supports materialisation-based parallel datalog reasoning and SPARQL query answering. It is developed and maintained at the University of Oxford and is available for download[1] under an academic licence. It is available on Linux, Mac OS X, Solaris, and Windows. It can be integrated as a library into C++, Java, and Python applications using an efficient native API; moreover, it can also be used as a standalone server accessible via a SPARQL endpoint. These versatile modes of use, combined with the very efficient storage and reasoning capabilities that we describe next, make RDFox suitable for a wide range of Semantic Web application scenarios.

RDFox supports datalog reasoning over RDF data using several novel datalog evaluation algorithms. To compute datalog materialisations, RDFox uses a shared-memory parallel algorithm that evenly distributes workload to threads by partitioning the reasoning task into many small, independent subtasks [15]. To support changes to the input data without recomputing materialisations from scratch, RDFox employs a novel incremental reasoning algorithm [14] that reduces the overall work by identifying early on whether a fact should be deleted or kept as a consequences of the update.

Many Semantic Web applications use the owl:sameAs property to state equalities between resources, which should be taken into account during materialisation. With many equality statements, however, this can significantly increase the memory consumption and degrade the overall reasoning performance [11]. *Rewriting* is a well-known technique for efficient equality reasoning [2,20], where equal resources are substituted during reasoning by a common representative. The result of this technique is called an *r-materialisation*, and it consists of a mapping between resources and their representatives and a dataset over the representatives. The correct computation of r-materialisations is not straightforward even on a single thread, since equalities derived during reasoning may trigger changes of representatives, which may require the deletion of outdated facts as well as changes to the datalog rules. RDFox employs a novel algorithm that seamlessly incorporates the rewriting technique into the parallel materialisation processes without sacrificing the benefits of parallelisation [17]. Moreover, updating r-materialisations incrementally is highly nontrivial, and the main difficulties stem from the fact that retraction of equalities between resources requires the reevaluation of all facts containing the representative of these resources. RDFox provides support for the incremental update of *r-materialisations* using a novel algorithm that was proved very efficient for small to medium-sized updates [16].

[1] http://www.rdfox.org/

To the best of our knowledge, RDFox is the only system that supports incremental update of r-materialisations.

To ensure scalability of data storage and access, RDFox uses a novel, efficient RDF storage scheme. It stores RDF triples in RAM, which is much faster than disk-based schemes, particularly for random access. The storage scheme uses compact data structures that can store hundreds of millions of triples on commodity PCs, and tens of billions of triples on high-end servers. The storage scheme comes in two variants: an economical version that can store up to four billion triples, and a more scalable version that can store more triples at the expense of using more bytes per triple. The storage scheme has configurable indexes that support efficient data access. All indexes support highly scalable, 'almost' lock-free parallel updates, which is critical for the performance of parallel reasoning. A particular challenge is to ensure eager elimination of duplicate triples, which is important for both the performance and correctness of reasoning.

In our previous work, we have demonstrated the scalability of RDFox on mid-range servers. In particular, RDFox can store 1.5 G triples in 52 G of RAM [15]; on 16 physical cores our parallel materialisation algorithms achieve reasoning speedup over the single-threaded version of up to 13.9 [15,17]; even in the single-threaded mode, RDFox often outperforms state of the art solutions based on relational and columnar databases [15]; and the (r-)materialisation can be efficiently updated for small to medium-sized updates [14,16]. To test the limits of the storage and reasoning scalability of RDFox, in this paper we shift our focus to high-end servers and present the results of a performance evaluation on a SPARC T5 with TB of RAM and 128 physical cores powering 1024 virtual cores via hyperthreading. Our evaluation shows very promising results: RDFox achieved speedups of up to 87 times (with 1024 threads) over the single-threaded version, storage of up to 9.2 billion triples, memory usage as low as 36.9 bytes per triple, importation rates of up to 1 million triples per second, and reasoning rates of up to 6.1 million triples per second.

The rest of the paper is structured as follows. In Section 2 we discuss the features and the different ways of accessing RDFox. In Section 3 we discuss in detail the architecture of RDFox. In Section 4 we demonstrate by means of an example the key ideas behind the algorithms used in RDFox. Finally, in Section 5 we describe the results of our performance evaluation.

2 Features, APIs, and Use Cases of RDFox

We now discuss the features of RDFox, the possible ways in which the system can be integrated into applications, and practical scenarios in which it has been employed.

RDFox Features. In RDFox, a *data store* is the basic RDF data management unit. Each data store is associated with a type that determines the data storage and indexing strategies. In any application, one can instantiate an arbitrary number of data stores, each of which provides the following functionality.

- Triples can be added to a data store in one of three possible ways: they can be *imported*, after which they are available for querying and reasoning, or they can be scheduled for *incremental addition* or *incremental deletion*, which makes them available for incremental reasoning. In each case, triples can be added programatically, read from an RDF 1.1 Turtle file, or extracted from an OWL 2 ontology.
- Analogously, a data store can import, or schedule for addition or deletion a set of datalog rules. Rules can be represented programmatically, read from a file in a custom *RDF datalog* format, or extracted from the OWL RL fragment of an ontology.
- A data store can answer SPARQL queries. Currently, RDFox supports most, but not all of SPARQL 1.1; in particular, we are still working on supporting aggregate queries and property paths.
- A data store can materialise the available triples with respect to the current set of rules. Reasoning can be carried out with or without optimised equality handling; the data store ensures that observable results in both cases are identical. The materialisation becomes available for querying immediately after reasoning has completed.
- One can incrementally update the materialisation according to the triples and rules scheduled for addition and/or deletion. RDFox does not support transactional updates; thus, the results of SPARQL queries are uniquely defined only after incremental update terminates.
- The triples in a store can be exported into a Turtle file, and the rules in a store can be exported into an RDF datalog file.
- The entire contents of a data store can be saved into a binary file, which can later be loaded to completely restore the state of the data store.

RDFox APIs. The core of RDFox is written in C++, but the system supports a number of APIs that enable integration with different kinds of applications.

- RDFox can be loaded as a C++ library, and all of its functionality can be easily accessed using a rich C++ API.
- RDFox can be accessed as a native library from Java and Python using suitable APIs. The Java and Python APIs act as a façade over the C++ API and provide access to the commonly used functionality.
- RDFox also supports a simple scripting language that supports command-line interaction with the system. This mode of interaction is particularly useful for ad hoc tests of the system's functionality, as well as for exploring the data and the effects of various operations.
- RDFox can be started in a server mode, providing access via a SPARQL endpoint. The endpoint currently supports only query answering, but our future plans include support for SPARQL updates.

Use Cases. RDFox is used in various industrial prototype systems, some of which we describe next.

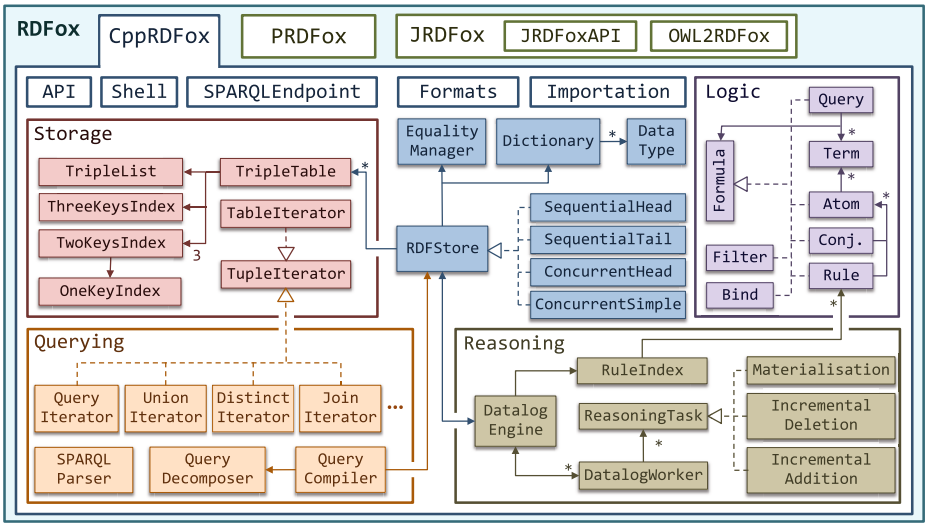

Fig. 1. RDFox Architecture

- **Statoil ASA** is a Norwegian multinational oil and gas company, and they are currently using RDFox as part of a large-scale Semantic Web application that facilitates the integration and analysis of oil production and geological survey data.
- **Électricité de France (EDF)** is a French electric utility company, and they are currently using RDFox to manage and analyse information about their electricity distribution network.
- **Kaiser Permanente** is a US health care consortium, and they are currently using RDFox to analyse patient data records.

3 System Architecture

The architecture of RDFox is summarised in Figure 1. It consists of a C++ module `CppRDFox`, a Java module `JRDFox`, and a Python module `PRDFox`. `CppRDFox` implements most of the functionality of RDFox and we discuss its structure in the rest of this section. `JRDFox` and `PRDFox` implement the Java and Python APIs, respectively, for the core functionality of `CppRDFox`. Moreover, `OWL2RDFox` uses the OWL API to load OWL 2 ontologies, extract their OWL 2 RL part, and translate the result into datalog; we discuss this translation in Section 4.2.

3.1 An Overview of `CppRDFox`

An `RDFStore` is the central concept of `CppRDFox` responsible for the storage, materialisation, and access of RDF data. There are several `RDFStore` variants,

each differing in its storage capacity, indexing scheme, support for parallel operation, and support for owl:sameAs reasoning; some of these variants are shown in Figure 1. Crucially, *sequential* store variants support only single-threaded access but without any synchronisation overhead, whereas *parallel* store variants support multi-threaded access and parallel materialisation. An `RDFStore` relies on the `TripleTable` from the `Storage` package for efficient storage of RDF triples and on the `DatalogEngine` from the `Reasoning` package for efficient datalog reasoning. Following common practice in RDF stores, RDF resources are encoded as integers to support efficient data storage and access; the `Dictionary` component manages this encoding. Finally, the `EqualityManager` component records the representatives for equal individuals.

The `Logic` package models well-known concepts from first-order logic, such as triple patterns, rules, and queries. The classes in this package provide the basis for the C++ API as they are used to represent the input to most components of the system. In this way, applications can interact with RDFox programmatically, and not just by representing triples and rules textually, which can eliminate a significant source of overhead.

The `Formats` package provides a pluggable architecture for various input/output formats supported by RDFox. At present, Turtle 1.1 and RDF datalog are fully supported, and support for RDF/XML is being finalised. The `Importation` package implements parallel importation of files. The `Querying` package handles the evaluation of SPARQL queries. The `Reasoning` package implements materialisation and incremental update algorithms. Finally, the `API` package provides a C façade over the native C++ API; the `Shell` package implements the aforementioned scripting language; and the `SPARQLEndPoint` package implements a SPARQL endpoint.

3.2 Components of `CppRDFox`

We next describe the main components of `CppRDFox` in more detail.

Dictionary. As is common in practice [4], the `Dictionary` component of RDFox encodes each RDF resource using a unique integer ID. These IDs are allocated sequentially starting from one and are thus 'small', which allows us to use the IDs as array indexes in various parts of the `RDFStore` component. A `Dictionary` uses various `DataType` implementations to handle different kinds of RDF literals. Each `DataType` instance is responsible for converting a lexical literal representation into a binary one, which involves literal normalisation. For example, integers 007 and 7 are both represented as the same binary value 7. While such an approach deviates slightly from the RDF specification, it allows us to store literals efficiently.

Storage. The `RDFStore` component stores RDF triples using a `TripleTable` component. Each `TripleTable` consists of a `TripleList` that stores the actual triples, as well as a number of indexes that support efficient iteration over subsets of the triples stored in the `TripleList`. A `TripleList` stores RDF triples as a two-dimensional array with six columns: the first three columns hold the IDs

of the subject, predicate, and object of a triple, while the latter three columns are used for indexing. In particular, the triples in the `TripleList` are organised in three linked lists, each of which is grouped by subject, predicate, and object, respectively; thus, the last three columns in the `TripleList` provide the next pointers in the respective lists. These linked lists are used to efficiently iterate over triples matching a combination of subject, predicate, and object. The grouping is fully configurable (at compile time), and it is chosen to support efficient answering of *triple patterns* of the form (t_s, t_p, t_o), where each of t_s, t_p, and t_o is either a variable or a resource identifier; the iteration is achieved via the `TableIterator` component. The `ThreeKeysIndex` implements a hash table over all triples in the `TripleList`, thus allowing for efficient duplicate elimination. We next discuss how RDFox answers different kinds of triple patterns.

Triple patterns not containing individuals are answered by sequentially scanning the `TripleList` and skipping over triples that do not match the pattern; for example, (x, y, x) is answered by skipping over triples whose subject and object differ. Triple patterns containing only individuals (i.e., variable-free patterns) are answered by a lookup into the `ThreeKeysIndex`. For triple patterns with one or two variables, a `TripleTable` relies on three `TwoKeysIndex` components, one for each of the components subject, predicate, and object, each maintaining one of the three triple lists. Each `TwoKeysIndex` contains a `OneKeyIndex` that, given a resource ID, locates the first triple in the relevant list with the given ID. Triple patterns containing just one resource ID are thus answered by iterating over the relevant list and possibly skipping over triples not matching the pattern. Since IDs are 'small', `OneKeyIndex` is implemented as an array, which supports efficient update and access. Triple patterns containing two resources are answered in two possible ways, depending on the configuration of the relevant `TwoKeysIndex`.

In the *simple* configuration, the `TwoKeysIndex` works similarly to when just one component is specified: it iterates over all triples containing the first component, and skips over the triples not matching the rest of the pattern. For example, if the triple pattern is (s, p, z), the `TwoKeysIndex` maintaining the triple list for s uses its `OneKeyIndex` to identify all triples containing s in the subject component, and skips over all triples not containing p in the predicate component. The benefit of this indexing scheme is simplicity of updates and low memory use, but the drawback is that answering certain triple patterns can be inefficient due to skipping.

In the *complex* configuration, the `TwoKeysIndex` maintains its relevant triple list grouped by an additional component, and it uses a hash table to efficiently locate the relevant sublist. For example, the `TwoKeysIndex` can keep the triples organised by first subject and then predicate; then, the triple pattern (s, p, z) is answered by querying the hash table by (s, p) to find the triples that contain s and p in their subject and predicate components, respectively, and by iterating over the triples in the list until the predicate component becomes different form p. Triple patterns of the form (s, y, o) are answered as in the case of a simple `TwoKeysIndex`. This indexing scheme offers more efficient triple retrieval with

no skipping, but comes at the expense of using more memory and more complex updates.

Currently, RDFox supports two kinds of indexing schemes. The *simple indexing scheme* employs a simple `TwoKeysIndex` for each of the three triple lists. In contrast, the *complex indexing scheme* employs a simple `TwoKeysIndex` for the predicate triple list, a complex `TwoKeysIndex` for the subject list grouped by predicate, and a complex `TwoKeysIndex` for the object list grouped by predicate. Hence, the complex indexing scheme can answer directly (i.e. without skipping) all triple patterns except for patterns of the form (s, y, o), which are delegated to the `TwoKeysIndex` responsible for the subject triple list.

Alternative data indexing schemes, such as the one used in RDF-3X [19], maintain sorted indexes, which allows for high degrees of data compression as well as answering many queries using very efficient merge joins. However, the maintenance of such indexes can be very costly and difficult to parallelise, and so such indexing schemes can be inefficient in scenarios where data changes continuously and in parallel, as is the case of parallel datalog materialisation. In contrast, the data indexing scheme employed by RDFox supports efficient, low-contention parallel maintenance, and is thus highly suitable for parallel datalog materialisation (for full details see [15]).

Querying. The querying package is responsible for SPARQL query answering. To evaluate a query, one can construct a `Query` object either programmatically or by parsing a SPARQL 1.1 query using the `SPARQLParser` component. The `SPARQLCompiler` component converts a `Query` object into an `TupleIterator` component that provides iteration over the answers. The `SPARQLCompiler` can optionally be configured to use a `QueryDecomposer` to produce query evaluation plans based on the extensive theory of queries of bounded treewidth [5]. Such query plans have proved critical to answering certain hard queries, but further investigation is required to make them useful in general. RDFox contains many different `TupleIterator` variants, each implementing specific SPARQL constructs. For example, `TableIterator` supports iteration over SPARQL triple patterns, `DistinctIterator` implements the "DISTINCT" construct of SPARQL, `UnionIterator` implements the "UNION" construct, and `QueryIterator` represents entire queries.

Reasoning. The reasoning package implements datalog materialisation and incremental updates. The `DatalogEngine` component organises the reasoning process. To support parallel reasoning, `DatalogEngine` uses `DatalogWorker` components, each of which can execute on one thread one of the reasoning tasks: `Materialisation`, `IncrementalDeletion`, and `IncrementalAddition`. Note that incremental reasoning is split into two tasks since incremental deletion has to be performed before incremental addition and the latter task cannot start before the former task finishes. Each task can work with or without rewriting of owl:sameAs. The datalog program loaded into RDFox is stored in a `RuleIndex` object, which, given a triple, can efficiently identify the rules for which the specified triple matches a body triple pattern.

Importation. For parallel `RDFStore` variants, the `Importation` package can be used to import multiple files in parallel. Requests are handled by the `ImportEngine`, which initialises a configurable number of `ImportWorker` components, and a collection of `ImportTask` components, one for each file to import. Each `ImportWorker` then iteratively extracts and executes an `ImportTask` until all tasks have been processed and so all files have been imported.

4 Datalog Reasoning

In this section, we present an overview of the algorithms that RDFox uses to efficiently compute and update datalog materialisations of RDF data. These algorithms are also applicable to the less expressive but more widely used OWL 2 RL language. Towards this goal, we first discuss how datalog can be integrated with RDF, then we discuss two different ways of supporting OWL 2 RL reasoning using datalog, and finally we demonstrate the key ideas behind our reasoning algorithms by means of an example.

4.1 RDF Datalog

A *term* is a variable or an RDF resource. A *triple pattern* is a triple (t_s, t_p, t_o), where t_s, t_p, and t_o are terms. An *(RDF) rule* r has the form (1)

$$H \leftarrow B_1 \wedge \cdots \wedge B_k, \tag{1}$$

where H is the *head* triple pattern, and each B_i, $1 \leq i \leq k$, is a *body* triple pattern. A *program* is a finite set of rules. A rule r' is an *instance* of a rule r if r' can be obtained from r by uniformly replacing the variables in r by RDF resources.

4.2 Common Approaches to OWL 2 RL Reasoning via Datalog

There are two main approaches to OWL 2 RL reasoning in datalog. In the first approach, the data and the ontology axioms are encoded as triples, and they are interpreted using the fixed set of rules from the OWL 2 RL specification [13, Section 4.3]. For example, consider the data triple $(\mathsf{peter}, \mathsf{type}, \mathsf{Teacher})$ (stating that peter is a Teacher) and the ontological triple $(\mathsf{Teacher}, \mathsf{subClassOf}, \mathsf{Person})$ (stating that the Teacher class is a subclass of the Person class). Then triple $(\mathsf{peter}, \mathsf{type}, \mathsf{Person})$ follows from these two triples, and it can be derived using the following rule from the OWL 2 RL specification:

$$(x, \mathsf{type}, y_2) \leftarrow (x, \mathsf{type}, y_1) \wedge (y_1, \mathsf{subClassOf}, y_2) \tag{2}$$

Using a fixed rule set may seem appealing due to its simplicity, but it can be inefficient. First, the fixed rules must match both data and ontological triples, and so they often contain many joins. Second, the rule set promotes considerable redundancy. For example, if we add $(\mathsf{Person}, \mathsf{subClassOf}, \mathsf{Mammal})$, due to the

transitivity of subClassOf we derive (Teacher, subClassOf, Mammal); but then, rule (2) derives (peter, type, Mammal) twice. In practice, such redundant derivations can incur significant overhead.

In the second approach, an OWL 2 RL ontology is translated into datalog rules that derive the same data triples. Our example ontology thus produces the following rules:

$$(x, \mathsf{type}, \mathsf{Person}) \leftarrow (x, \mathsf{type}, \mathsf{Teacher}) \tag{3}$$

$$(x, \mathsf{type}, \mathsf{Teacher}) \leftarrow (x, \mathsf{type}, \mathsf{Mammal}) \tag{4}$$

These rules also derive the triples (peter, type, Teacher) and (peter, type, Mammal); however, each rule contains only one body triple pattern and so it can be evaluated more efficiently. Furthermore, all data triples are derived only once.

RDFox can handle arbitrary datalog programs and so it can support both approaches to OWL 2 RL reasoning. For efficiency, we use the second approach in our evaluation.

4.3 Computing Datalog Materialisations

To compute a datalog materialisation, we must exhaustively apply all rules to the dataset until no new triples can be derived. We demonstrate this using an example dataset E and datalog program Σ. The dataset E consists of triples (E1)–(E3), which we call *explicit triples*, and the datalog program Σ consists of the rules (R1)–(R4), which correspond to typical OWL 2 RL axioms.

$$(\mathsf{john}, \mathsf{teach}, \mathsf{math}) \tag{E1}$$

$$(\mathsf{john}, \mathsf{teach}, \mathsf{phys}) \tag{E2}$$

$$(\mathsf{peter}, \mathsf{teach}, \mathsf{math}) \tag{E3}$$

$$(x, \mathsf{type}, \mathsf{Teacher}) \leftarrow (x, \mathsf{type}, \mathsf{Person}) \wedge (x, \mathsf{teach}, y) \wedge (y, \mathsf{type}, \mathsf{Course}) \tag{R1}$$

$$(x, \mathsf{type}, \mathsf{Person}) \leftarrow (x, \mathsf{type}, \mathsf{Teacher}) \tag{R2}$$

$$(x, \mathsf{type}, \mathsf{Person}) \leftarrow (x, \mathsf{teach}, y) \tag{R3}$$

$$(y, \mathsf{type}, \mathsf{Course}) \leftarrow (x, \mathsf{teach}, y) \tag{R4}$$

Rules (R1) and (R2) capture the OWL 2 RL consequences of axiom

$$\mathit{EquivalentClasses}(\mathsf{Teacher}\ \mathit{ObjectIntersectionOf}(\mathsf{Person}\ \mathit{ObjectSomeValuesFrom}(\mathsf{teach}\ \mathsf{Course})))$$

and rules (R3) and (R4) state that classes Person and Course are the domain and the range, respectively, of property teach. The materialisation I of E w.r.t. program Σ extends E with triples (I1)–(I6), which we call *implicit*.

$$(\mathsf{john}, \mathsf{type}, \mathsf{Person}) \tag{I1}$$

$$(\mathsf{math}, \mathsf{type}, \mathsf{Course}) \tag{I2}$$

$$(\mathsf{phys}, \mathsf{type}, \mathsf{Course}) \tag{I3}$$

$$(\mathsf{peter}, \mathsf{type}, \mathsf{Person}) \tag{I4}$$

$$(\mathsf{john}, \mathsf{type}, \mathsf{Teacher}) \tag{I5}$$

$$(\mathsf{peter}, \mathsf{type}, \mathsf{Teacher}) \tag{I6}$$

In particular, applying rule (R3) to either (E1) or (E2) produces (I1); applying rule (R4) to either (E1) or (E3) produces (I2); applying rule (R4) to (E2) produces (I3); and applying rule (R3) to (E3) produces (I4). Moreover, applying rule (R1) to (I1), (I2), and (E1) produces (I5), and applying rule (R1) to (I2), (I4), and (E3) produces (I6). At this point, applying rules (R1)–(R4) to I derives no new triples, so materialisation finishes.

A naïve materialisation approach is to repeatedly apply the rules to the available triples as long as any fresh triples are derived. Using such an approach, we would compute the materialisation as follows: we first apply rules (R1)–(R4) to triples (E1)–(E3) to derive (I1)–(I4); then, we apply (R1)–(R4) again to (E1)–(E3) and (I1)–(I4) to derive (I5)–(I6). This, however, would be very inefficient as in the second application of the rules we would again derive (I1)–(I4) only to discover that these triples have already been derived in the first iteration. Such redundant derivations would pose a considerable source of inefficiency, so such naïve approaches are unsuitable for practical use.

To prevent redundant derivations from the previous paragraph, RDFox uses a novel materialisation algorithm [15] that captures the idea behind the well-known *seminaïve* materialisation approach [1]. The algorithm avoids redundant derivations by considering only rule instances with at least one freshly derived body triple. Roughly speaking, each thread in RDFox extracts an unprocessed triple from the current dataset, matches the triple in all possible ways to triple patterns in a rule body, extends each such match to a rule instance by querying the already processed triples, and adds the instantiated rule head to the dataset. Whenever a thread adds a new triple to the dataset, it notifies all threads that work is available. When there are no more triples to be extracted, the thread goes to sleep if there are still active threads; otherwise, it notifies all threads (all of which must be sleeping) that the materialisation has been completed.

This algorithm breaks down the reasoning process into as many subtasks as there are triples in the materialisation, and these subtasks are dynamically assigned to threads without any need for scheduling or any form of explicit load balancing. Consequently, in all but pathological cases, the algorithm distributes the work to threads evenly. This is in contrast to known approaches that parallelise materialisation by statically assigning either rules or rule instances to threads and are thus often susceptible to data skew.

We next show using our example how our algorithm avoids repeating derivations. A thread first extracts (E1) to derive (I1) and (I2); then, it extracts (E2) to derive (I3); and it extracts (E3) to derive (I4). When the thread extracts (I1), it matches the triple to the first body triple pattern of (R1), but this fails to produce an instance of (R1): although (I2) is available, it has not been processed yet. A thread then extracts (I2) and matches it to the third body triple pattern of (R1); the rule can now be instantiated using only processed triples to derive (I5); thus, the rule instance of (R1) that derives (I5) is considered only once.

4.4 Updating Datalog Materialisations

Instead of recomputing the materialisation from scratch when some of the explicit triples change, it is often desirable to update the materialisation *incrementally*—that is, with as little work as possible. Different such approaches have been considered: some require collecting information during the initial materialisation, whereas others require no extra information; please refer to [14] for an overview. We found the latter approaches more suitable for main-memory systems such as RDFox, where low memory consumption is critical. Moreover, adding explicit triples is generally easy because one can just restart the initial materialisation process, so in the rest of this section we focus on triple deletion.

Assume that we want to delete triple (E1) from our running example. Then, we must identify all triples that can be derived directly or indirectly using (E1), and then determine whether these triples have alternative derivations or need to be deleted as well. The *delete/rederive* (DRed) algorithm [8] is a well-known algorithm that follows this approach, and it proceeds as follows. First, in the *overdeletion* stage, the algorithm identifies all triples that have (directly or indirectly) been derived from (E1). Concretely, the algorithm applies rules (R1), (R3), and (R4) to I while matching at least one body triple pattern to (E1); consequently, triples (I5), (I1), and (I2) are deleted as well. By applying this process further, all implicit triples except (I3) are deleted. All of these triples, however, have an alternative derivation from (E2)–(E3); hence, in the *rederivation* stage, DRed reintroduces all such triples by applying the rules to the 'surviving' explicit triples. As this example demonstrates, the algorithm can potentially delete a large portion of the materialised dataset just to reintroduce it later, which can be inefficient. This problem is particularly acute when triples have many alternative derivations, which is often the case in Semantic Web applications.

DRed propagates the deletion of a triple regardless of whether the triple has alternative derivations or not. As a remedy, RDFox uses the *backward/forward* (B/F) algorithm [14]: before deleting a triple, the algorithm checks using a combination of backward and forward chaining whether an alternative derivation exists, and it deletes a triple only if that is not the case. Consider again the deletion of (E1). The B/F algorithm first tries to identify an alternative derivation by matching (E1) to the head of a rule; as this cannot be done, the algorithm deletes (E1). It then examines the direct consequences of (E1) in exactly the same ways as in DRed, and thus identifies (I1), (I2), and (I5) as the direct consequences of (E1). For each of these triples, B/F next tries to identify an alternative proof. In particular, the algorithm determines that (I1) is derivable using rule (R3) the the explicit 'surviving' triple (E2), and so it will not delete (I1); this will prevent the algorithm from further considering the consequences of (I1), which improves the overall performance of the algorithm. The checking of alternative proofs is more involved due to the need to ensure termination of backward chaining in the presence of recursive rules; please refer to [14] for details.

4.5 Handling owl:sameAs Using Rewriting

The owl:sameAs property states that two resources are equal: if (a, owl:sameAs, b) holds, then a and b can be used interchangeably. The semantics of owl:sameAs can be captured using the following rules:

$$(x_i, \mathsf{owl{:}sameAs}, x_i) \leftarrow (x_1, x_2, x_3) \quad \text{for } 1 \leq i \leq 3 \qquad \text{(EQ1)}$$

$$(x'_1, x_2, x_3) \leftarrow (x_1, x_2, x_3) \wedge (x_1, \mathsf{owl{:}sameAs}, x'_1) \qquad \text{(EQ2)}$$

$$(x_1, x'_2, x_3) \leftarrow (x_1, x_2, x_3) \wedge (x_2, \mathsf{owl{:}sameAs}, x'_2) \qquad \text{(EQ3)}$$

$$(x_1, x_2, x'_3) \leftarrow (x_1, x_2, x_3) \wedge (x'_3, \mathsf{owl{:}sameAs}, x_3) \qquad \text{(EQ4)}$$

Rules (EQ2)–(EQ4) 'copy' triples between equal resources, which can adversely impact memory consumption [11] and reasoning performance [17].

Rewriting is an optimisation widely used by datalog materialisation algorithms to efficiently handle owl:sameAs reasoning. The idea is to replace all equal individuals by a common representative. The result of this technique is called an r-materialisation and it consists of a mapping between resources and their representatives and a dataset over the representatives. Consider, for example, the dataset E and the program Σ_{eq} obtained by extending Σ with rule (R5) that makes property teach inverse-functional.

$$(x, \mathsf{owl{:}sameAs}, y) \leftarrow (x, \mathsf{teach}, z) \wedge (y, \mathsf{teach}, z) \qquad \text{(R5)}$$

By applying (R5) to (E1) and (E3), we determine that john and peter are equal. To apply rewriting, we choose one of the two resources as the representative of the other; for example, let us choose john as the representative of peter. The r-materialisation of E w.r.t. Σ_{eq} then contains triples (I1)–(I3) and (I5), which are obtained from (I1)–(I6) by replacing peter with john.

The parallel r-materialisation algorithm of RDFox [17] extends the algorithm from Section 4.3. In the extended algorithm, each thread can perform one of three possible actions. First, a thread can extract and process a triple in the dataset; if the triple is outdated (i.e., it contains a resource for which a different representative has been defined), then the triple is deleted and its updated version is added to the dataset; if the triple is of the form (s, owl:sameAs, o) with $\mathsf{s} \neq \mathsf{o}$, then the thread identifies one resource as the representative of the other and adds the outdated resource to a special list; and in all other cases the thread applies rules to the triple as in the original materialisation algorithm. Second, a thread can extract an outdated resource c, delete each triple containing c and add its updated version to the dataset, and update all rules containing c. Third, a thread can evaluate a rule that was updated in the previous case.

Updating r-materialisations is nontrivial. First, deleting an equality may actually require *adding* triples. Consider again the dataset E, the program Σ_{eq}, and their r-materialisation computed as explained above, and assume again that we delete triple (E1). After the deletion, john is no longer equal to peter, and so (I4) and (I6) must be added to the r-materialisation; thus, the r-materialisation after deletion contains (I1)–(I6). Second, if we delete an equality containing a

resource c, we must reevaluate each triple that contains a resource that c represents. RDFox supports incremental r-materialisation updates using a novel algorithm that has been shown to be very efficient for small to medium-sized updates [16].

5 Evaluation

We tested RDFox on an Oracle SPARC T5-8 server. The system has 8 SPARC V9 processors with 16 physical cores per processor, each supporting 8 threads via hyperthreading; thus, the system supports 128 physical and 1024 virtual threads in total. The processors run at 3.6GHz, and each processor has 16KB of instruction cache, 16KB of data cache, 128KB of L2 cache, and 8MB of L3 cache. The system has 4TB of DDR3 memory and is running Solaris 11.1.

Test Data. We now describe the datasets that we used to evaluate RDFox; all datasets are available online.[2] The Lehigh University Benchmark (**LUBM**) [7] is a widely used synthetic benchmark for RDF systems. The LUBM ontology describes the university domain, and the data is generated by specifying a number of universities, with each university contributing about 100k triples. We used the LUBM-50K dataset with 50,000 universities, which in compressed form was 37 GB. For the rules, we extracted the *lower bound* from the LUBM ontology—that is, we identified the OWL 2 RL part of the ontology and converted it into an RDF datalog program using the transformation by [6]; we call the resulting program LUBM_L. **Claros** is a cultural database cataloguing archaeologic artefacts. For the rules, we extracted the lower bound as above, but, to push the limits of RDFox, we extended the lower bound with some manually generated rules; we call the resulting datalog program Claros_{LE}. **DBpedia** represents structured data extracted from Wikipedia. As in the case of Claros, we extracted the lower bound and extended it with several challenging rules; we call the resulting program DBpedia_{LE}.

Materialisation Tests. Table 1 summarises the results of our materialisation tests. For each dataset, we measured the time needed to import the data (shown under 'import') without any materialisation, and the memory usage per triple ('B/trp') and the number of triples ('Triples') after import ('aft imp'). The number of threads using during import was limited by the number of files storing the data; thus, we used just one thread for Claros and DBpedia, and 11 threads for LUBM-50K. We then computed the materialisation of the dataset while varying the number of threads. For each test, we show the overall time in seconds, as well as the speedup over using just one thread. For each dataset, we show the memory usage per triple ('B/trp') and the number of triples ('Triples') after materialisation ('aft mat'). Finally, for each dataset we show the maximum rates in triples/second achieved during import ('import rate') and materialisation ('mat. rate'); the former is the number of triples before materialisation divided by the import time, and the latter is the difference in the numbers of

[2] https://krr-nas.cs.ox.ac.uk/2015/ISWC/index.html

Table 1. Summarisation of the conducted tests

	LUBM-50K		Claros		DBpedia	
Threads	sec	speedup	sec	speedup	sec	speedup
import	6.8k	—	168	—	952	—
1	27.0k	1.0x	10.0k	1.0x	31.2k	1.0x
16	1.7k	15.7x	906.0	11.0x	3.0k	10.4x
32	1.1k	24.0x	583.3	17.1x	1.8k	17.5x
48	920.7	29.3x	450.8	22.2x	2.0k	16.0x
64	721.2	37.4x	374.9	26.7x	1.2k	25.8x
80	523.6	51.5x	384.1	26.0x	1.2k	26.7x
96	442.4	60.9x	364.3	27.4x	825	37.8x
112	400.6	67.3x	331.4	30.2x	1.3k	24.3x
128	387.4	69.6x	225.7	44.3x	697.9	44.7x
256	—	—	226.1	44.2x	684.0	45.7x
384	—	—	189.1	52.9x	546.2	57.2x
512	—	—	153.5	65.1x	431.8	72.3x
640	—	—	140.5	71.2x	393.4	79.4x
768	—	—	130.4	76.7x	366.2	85.3x
896	—	—	127.0	78.8x	364.9	86.6x
1024	—	—	124.9	80.1x	358.8	87.0x
size	B/trp	Triples	B/trp	Triples	B/trp	Triples
aft imp	124.1	6.7G	80.5	18.8M	58.4	112.7M
aft mat	101.0	9.2G	36.9	539.2M	39.0	1.5G
import rate	1.0M		112k		120k	
mat. rate	6.1M		4.2M		4.0M	

triples after and before materialisation divided by materialisation time. Note that we could use just one thread while importing Claros and DBPedia, so the import rate is primarily limited by the speed of our Turtle parser. LUBM does not use owl:sameAs, so for we used the `RDFStore` without support for rewriting; in contrast, for Claros and DBpedia we used the `RDFStore` with rewriting support. In all cases we used the complex `RDFStore` variant, as it provides more efficient query evaluation.

We were unable to complete tests on LUBM-50K with more than 128 threads. As we discussed in [15], to reduce thread interference, each reasoning thread uses additional memory that depends on the number of triples; hence, RDFox exhausted the available memory with more than 128 threads. Optimising memory consumption with many threads is an important topic for our future work.

Memory Usage. For LUBM-50K, we used a version of `RDFStore` that uses 8-byte pointers and can thus store 2^{64} triples. We initialised the store to preallocate sufficient space for the target number of triples after materialisation. This eliminated the need for hash table resizing during import and materialisation, but due to a safety margin on the number of triples, RDFox used 101 bytes/triple, which is more than necessary: without preallocation, LUBM-50K could store the materialised dataset using 89.7 bytes/triple.

For Claros and DBpedia, we used a version of `RDFStore` that uses 4-byte pointers and can thus store 2^{32} triples. Claros was the smallest dataset; however, due to complex rules, materialisation increases the size of the data by a factor of 28. Due to this increase, the share of the size of the `Dictionary` drops from 30% before materialisation to 2%. The resulting dataset contains several large cliques of connected resources, so the variation in the number of different subject–property and object–property pair is low; this ensures that indexes are several

orders of magnitude smaller than the number of triples, so the entire dataset can be stored in only 36.9 bytes/triple. DBpedia is larger than Claros, but its rule set is similar in that it creates cliques of connected individuals. Hence, in the same way as in Claros, the dataset after materialisation can be stored very efficiently, using only 39 bytes/triple.

Reasoning Speedup. The target server supports only 128 physical cores, so 128 is the maximal possible speedup one can expect. As one can see from Table 1, RDFox achieved between 54% and 68% of the maximum, suggesting that our approach to parallelisation of reasoning is very effective. As one can see, parallelisation can be critical for dealing with large datasets and/or complex programs; for example, parallelisation reduces materialisation times on DBpedia from almost 9 hours to just under 6 minutes.

Materialisation in RDFox is a memory-bound task due to random index access, and so each core is susceptible to stalls due to CPU cache misses. However, as one can see from Table 1, hyperthreading seems to effectively compensate for this: on both Claros and DBpedia it roughly doubles the materialisation speed. Please refer to [15] for a more in-depth discussion about the problems related to CPU cache locality.

Incremental Maintenance. To tests our incremental update algorithms, we extracted five subsets of 5,000 triples from the LUBM-50K dataset; for each subset, we measured the time used to update the materialisation after deletion. On average, RDFox could update the materialisation in 0.49s while removing 8525.8 triples in total; the fastest update took 0.42s and required deleting 8,451 triples, while the longest one took 0.6s and required deleting 8,520 triples.

6 Conclusion

In this paper, we have presented RDFox, a main-memory RDF store that supports parallel datalog reasoning. We have described the system architecture of RDFox together with its numerous APIs, its highly-efficient and flexible storage scheme, and its state-of-the-art datalog reasoning algorithms. Its open-source cross-platform implementation also allows for easy integration in a wide range of Semantic Web application scenarios. With storage capabilities of up-to 9.2 billion triples, datalog reasoning speeds of up-to 6.1 million triples per second, and parallel reasoning speedups of up to 87 times, RDFox opens new possibilities for data intensive applications requiring expressive and highly-scalable reasoning. With memory consumption as low as 36.9 bytes per triple, RDFox is also suitable for smaller-scale applications managing up to hundreds of millions of triples on commodity hardware. RDFox thus provides a unique combination of versatility, rich functionality, high performance, and scalability.

In our future work, we plan to extend the functionality of RDFox and improve its performance in a number of ways. Firstly, we plan to add support to all of SPARQL 1.1, and we are already working on an improved query answering algorithm. Secondly, we plan to add support for named graphs, which are becoming

increasingly popular in Semantic Web applications, as well as support for reasoning with non-monotonic negation. Finally, we are in the process of building a shared-nothing, distributed version of the system, which will allow for the efficient storing, querying, and reasoning with larger datasets using less powerful hardware.

Acknowledgments. We thank Hassan Chafi and Brian Whitney for providing access to the T5 system and their support on hardware and OS questions. This work was funded by the EPSRC projects MaSI^3, Score!, and DBOnto, and the FP7 project Optique.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. AW (1995)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. CUP (1998)
3. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: A family of scalable semantic repositories. Sem. Web **2**(1), 33–42 (2011)
4. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient SQL-based RDF querying scheme. In: Proc. VLDB, pp. 1216–1227 (2005)
5. Flum, J., Frick, M., Grohe, M.: Query Evaluation via Tree-Decompositions. Journal of the ACM **49**(6), 716–752 (2002)
6. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: WWW, pp. 48–57 (2003)
7. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. JWS **3**(2–3), 158–182 (2005)
8. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: Proc. SIGMOD, pp. 157–166 (1993)
9. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible SROIQ. In: Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2006), pp. 57–67 (2006)
10. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission (2004)
11. Kolovski, V., Wu, Z., Eadon, G.: Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 436–452. Springer, Heidelberg (2010)
12. Manola, F., Miller, E., McBride, B.: RDF Primer. W3C Rec. **10**, 1–107 (2004)
13. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language Profiles, 2nd edn. W3C Rec. (2012)
14. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Incremental update of datalog materialisation: the backward/forward algorithm. In: Proc. AAAI, pp. 1560–1568 (2015)
15. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In: Proc. AAAI, pp. 129–137 (2014)
16. Motik, B., Nenov, Y., Piro, R.E.F., Horrocks, I.: Combining rewriting and incremental materialisation maintenance for datalog programs with equality. In: Proc. (to appear 2015)

17. Motik, B., Nenov, Y., Piro, R.E.F., Horrocks, I.: Handling owl:sameAs via rewriting. In: Proc. AAAI, pp. 231–237 (2015)
18. Motik, B., Patel-Schneider, P.F., Parsia, B., Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., et al.: OWL 2 web ontology language: Structural specification and functional-style syntax. W3C Rec. **27**, 17 (2009)
19. Neumann, T., Weikum, G.: The RDF-3X Engine for Scalable Management of RDF Data. VLDB Journal **19**(1), 91–113 (2010)
20. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 7, pp. 371–443. Elsevier Science (2001)
21. SPARQL 1.1 Overview. W3C Recommendation (March 21, 2013)
22. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an inference engine for RDFS/OWL constructs and user-defined rules in oracle. In: ICDE, pp. 1239–1248 (2008)