Rinne, Mikko; Nuutila, Esko

User-Configurable Semantic Data Stream Reasoning Using SPARQL Update

Published in:
Journal on Data Semantics

DOI:
10.1007/s13740-017-0076-9

Published: 01/09/2017

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Published under the following license:
Unspecified

Please cite the original version:
Rinne, M., & Nuutila, E. (2017). User-Configurable Semantic Data Stream Reasoning Using SPARQL Update. *Journal on Data Semantics*, *6*(3), 125-138. https://doi.org/10.1007/s13740-017-0076-9

Journal on Data Semantics manuscript No.
(will be inserted by the editor)

User-Configurable Semantic Data Stream Reasoning using SPARQL Update

Mikko Rinne · **Esko Nuutila**

Received: 10 March 2016 / Accepted: 9 February 2017

Abstract Stream reasoning is one of the building blocks giving semantic web an advantage in the race for the real-time web. This paper demonstrates implementation of materialisation-based reasoning using an event processor supporting networks of specification-compliant SPARQL Update rules. Collections of rules coded in SPARQL leave the rule implementation exposed for selection and modification by the platform user using the same query language for both the queries and entailment rules. Observations on the differences of SPARQL and rule semantics are made. The entailment-category tests of the SPARQL 1.1 conformance test set are thoroughly reviewed. New rules are constructed to improve platform pass rate and the test results are measured. An *event-based memory handling* solution to the accumulation of data in stream processing scenarios through separation of static data (e.g. the ontology) from dynamic event data is presented and tested. This implementation extends the reasoning support available in an RDF stream processor from RDF(S) to ρdf, D*, P-entailment and OWL 2 RL. The performance of the INSTANS platform is measured using a well-known benchmark requiring reasoning, comparing complete sets of entailment rules against the necessary subset to complete each test. Performance is also compared to non-streaming SPARQL query processors with reasoning support.

Keywords Entailment · rule networks · SPARQL · stream reasoning

M. Rinne and E. Nuutila
Aalto University School of Science,
PO Box 15400, FI-00076 AALTO, FINLAND
E-mail: firstname.lastname@aalto.fi

1 Introduction

The support for finding logical consequences from a set of facts, rules and axioms – reasoning – has been built into the foundation of semantic web technologies. Many reasoners use first-order predicate logic, with the inference proceeding by forward- or backward-chaining [Singh and Karwayun(2010)]. Approach from the side of the query is known as *query rewriting* [Pérez-Urbina et al(2012), Imprialou et al(2012), Bischof et al(2014)] – fusion of the query with ontological information – and from the side of data as *materialisation* [Volz et al(2005)] – synthesis of new facts based on the application of a set of rules. Entailment regimes are defined by [Glimm and Ogbuji(2013)] as extensions of basic graph pattern matching of SPARQL semantics allowing additional RDF statements to be inferred from explicitly given assertions.

Core semantic web technologies RDF, SPARQL and OWL (Web Ontology Language) have reached second generation[1]. The complexity of a full-blown OWL inferencing tool, even for the more restricted OWL profiles like DL or Lite, or OWL 2 RL [W3C(2012)], can be considerable. Many of the entailment rules also produce a considerable overhead when used for materialisation[2], while bringing both limited and case-specific value to the reasoning process. Implementations occasionally ignore entailment rules[3] falling into this category. In the current situation:

1. Different entailment regimes partially overlap (illustrated in Figure 1)
2. Use of partially overlapping regimes in parallel may introduce problems and unexpected results.

[1] RDF and SPARQL as version 1.1, OWL as "OWL 2"
[2] quantitatively demonstrated in Section 6
[3] e.g. `rdfD2`, which states that all predicates are *properties* and `rdfs4`, which states that all subjects and objects are *resources*

3. Entailment rules may otherwise appear identical across regimes, but have different restrictions (e.g. limitations on data types) depending on the regime.
4. Full-scale entailment regime implementations working with materialisation produce impractical amounts of overhead for many tasks.
5. For many tasks only a small subset of the rules is required.
6. Due to subtle differences in e.g. the triggering conditions of otherwise identical rules an end-user needs to know exactly how entailment rules are implemented in order to correctly interpret results and compare them between platforms.
7. To repeat experiments (such as old test cases), the entailment rule implementation should be coupled with the experiment, not the platform.

The need for a combination of entailment regime subsets has been identified e.g. in [Polleres et al(2013)] in the context of linked open data. One solution to these challenges is to create a flexible reasoning framework, which enables the user of a platform to be fully informed of the exact implementation of the various entailment rules, to select and modify the rules as needed and to package the set of rules with a particular experiment.

INSTANS[4] [Rinne and Nuutila(2014)] is a continuously executing RDF stream processing platform based on the *Rete-algorithm* [Forgy(1982)] implementing SPARQL 1.1 Query [W3C(2013a)] and Update [W3C(2013b)] specifications. Originating from rule-based expert systems, Rete is targeted towards supporting a large number of parallel rules, or in this case, SPARQL queries. The platform allows to experiment with the materialisation of entailment triples using the same language as users of the platform would use to write queries - SPARQL. The following contributions to RDF stream reasoning are made:

- First compliance-tested entailment regime implementations using networks of specification-compliant SPARQL exclusively
- Memory handling in stream reasoning through controlled separation of static knowledge from event data
- Extension of RDF stream reasoning beyond RDF(S) entailment (ρdf, D*, P-entailment and OWL 2 RL)
- Performance comparison of an RDF stream processor with non-streaming processors using a benchmark with reasoning.

Even though stream reasoning has been discussed for a number of years (e.g. [Barbieri et al(2010a)]), no RDF stream processor with reasoning capability beyond simple RDF(S) entailment has been found at the time of writing. As no comparison platforms or streaming benchmarks with more comprehensive entailment regimes are available, we have used a non-streaming benchmark to enable performance testing and comparison with other platforms. The selected benchmark has event-like recurring data structures, which enable us to demonstrate event-based memory handling. Even though continuous query processing is an integral part of the INSTANS platform, the presented approach is not limited to stream processing. The accompanying rule libraries for different entailment regimes are hereby made openly available for any use or further development.

In Section 2 the history of relevant RDF, RDFS and OWL entailment regimes as well as reasoning in RDF stream processing platforms is reviewed. The incorporated entailment rules and their SPARQL implementations are explained in Section 3. The impact of the entailment rules on the results of the entailment tests in the SPARQL 1.1 test suite are reviewed in Section 4. The experimental setup for benchmarking INSTANS with reasoning is explained in Section 5. Performance results for full sets of entailment rules as well as optimised sets are documented in Section 6. Conclusions on the results are drawn in Section 7.

2 Background

The basic method of querying data with the SPARQL query language is to generate a query graph possibly including wild cards and find the RDF data graphs matching said query graph. Many W3C standards (e.g. RDF and OWL) provide semantic interpretations of RDF graphs allowing additional RDF statements to be inferred. These inferred statements can be generated using semantic entailment relations. Such a standard set of semantic web entailment relations is called an *entailment regime* [Glimm and Ogbuji(2013)]. All entailment regimes are defined as monotonic extensions of the simple RDF entailment regime, meaning that a semantic extension cannot cancel an entailment specified by a weaker regime.

In our work we have covered rule-based implementations under RDF 1.1 semantics of RDF entailment [W3C(2014b)], RDFS entailment [W3C(2014b)], and OWL 2 (RL profile [W3C(2012)]) as listed in [Glimm and Ogbuji(2013)]. Additionally we have experimented with the minimal subset of RDFS denoted ρdf [Muñoz et al(2007), Muñoz et al(2009)] as well as the completementary *D*-* and *P-entailments* specified by ter Horst [ter Horst(2004), ter Horst(2005a), ter Horst(2005b)], where the D*-entailment covers a subset of RDFS and P-entailment a subset of first-generation OWL. The documentation for SWCLOS2[5] (Semantic Web Common Lisp Object System) presents - in addition to the P-entailment - a set of

[4] Incremental eNgine for STANding Sparql, `http://instans.org`

[5] `https://github.com/SeijiKoide/SWCLOS/tree/master/Manual`

"additional entailment rules" as well as a set of "unsatisfiability rules", which only partially overlap OWL 2 RL, and have thus been included in the study as *SWCLOS2*.

An implementation of OWL2 RL with SPARQL has been previously demonstrated[6]. That solution makes use of SPIN[7] and built-in Jena[8] functions, whereas the solution presented in this paper consists exclusively of specification-compliant SPARQL. We were also unable to find either compliance or performance information for that SPARQL implementation. It targets *TopBraid Composer*[9], which also supports built-in OWL and RDFS reasoners.

Stream reasoning is used as a term for the combination of stream processing and reasoning [Valle et al(2013), Margara et al(2014)], sometimes focusing on agile, lightweight reasoning on rapidly changing information [Barbieri et al(2010a)]. C-SPARQL[10] [Barbieri et al(2010b), Barbieri et al(2010c)] is a SPARQL extension and an early platform for RDF stream processing. It is based on the *data stream processing* principle, where the first step of stream processing is to apply a so-called *stream-to-relation* (S2R) operator. The S2R operator extracts a *time window* of predefined duration or number of tuples from the stream. All subsequent query operations are scoped to the extracted window. In C-SPARQL there is no built-in method for preservation of state between subsequent windows. The stream reasoning approaches on C-SPARQL take advantage of this, defining expiration times for materialised triples based on window durations [Barbieri et al(2010a)]. The current version of C-SPARQL supports "simple RDF entailment".

In Sparkwave entailments have been implemented separately and denoted as the ε-*network* [Komazec and Cerri(2011), Komazec et al(2012)] to separate from α- and β-nodes found in the original Rete-algorithm [Forgy(1982)]. In [Komazec et al(2012)] it is concluded that "only rules `rdfs2`, `rdfs3`, `rdfs7` and `rdfs9` need consideration at runtime", but Sparkwave also adds three rules on `owl:inverseOf` and `owl:SymmetricProperty` to arrive at subsets of both RDFS and OWL entailments.

In *EP-SPARQL/ETALIS* [Anicic et al(2011)] an external library[11] is used to transform RDFS ontologies into Prolog rules and facts.

Outside the stream processing domain there are many established reasoning and SPARQL query processing platforms which can be used in multiple combinations. For the performance comparison we have chosen *Jena* as a well-known platform with an open and configurable materialisation-based reasoner and *Stardog*[12] as an example of a high-performance tool using query rewriting. Another strong candidate for the high-performance comparison platform would be *RDFox* [Nenov et al(2015)].

The typical way to support reasoning on an RDF processing platform is to make a set of reasoners available for selection together with a framework for implementing custom rules. E.g. Jena general purpose rule engine[13] has a custom rule syntax, while Stardog allows custom rule implementation in SWRL[14]. In these approaches the reasoners are a part of the platform and new platform versions may update rule implementation, which may impact results. While ready-made reasoners can often be complemented with custom rules, on platforms other than INSTANS we have not come across a documented method for an end-user to make a local copy of a ready-made reasoner implementation and remove rules, which are unnecessary for a particular task. In the approach presented in this paper, reasoners are implemented openly as SPARQL query networks in separate text files. An end-user can make a local copy, which can be freely edited with rules added, removed or modified. Updates to the reasoners can be made available, but they are not tied to platform versions and do not override the local copy of the end-user. No new syntax needs to be learned for specifying entailment rules, as everything is written in specification-compliant SPARQL. Any of the existing rules can be used as a template for creating a new rule.

3 Entailment rule implementation with SPARQL

Overlaps between the different sets of entailment rules considered in the study are illustrated in Figure 1. Two rules have been considered the same, when an identical input pattern materialises an identical output. When multiple triggering conditions have been listed under one rule (e.g. 7.(b) in ρdf [Muñoz et al(2009)]), the original has been split to multiple rules. "Rules" omitting a triggering condition have not been counted, as they have more commonality with *axiomatic triples* [W3C(2014b)]. Slight variations in input filtering, e.g. the condition for parameter p as $p \in U \cup B$ applied for some rules in D* and P-entailments, have not been considered a reason to separate the rules from otherwise identical rules in other regimes. This criteria yields 100 unique rules and 21 unique unsatisfiability conditions. As can be seen in Figure 1, a core set of 6 rules is common to RDFS, ρdf, D* and OWL 2 RL. Outside RDFS a significant overlap between OWL 2 RL and P-entailment

[6] `http://topbraid.org/spin/owlrl-all.html`
[7] `http://spinrdf.org/`
[8] `https://jena.apache.org/`
[9] `http://www.topquadrant.com/tools/IDE-topbraid-composer-maestro-edition/`
[10] `http://streamreasoning.org/resources/c-sparql`
[11] `http://www.swi-prolog.org/pldoc/package/semweb.html`
[12] `http://stardog.com/`
[13] `https://jena.apache.org/documentation/inference/`
[14] `http://www.w3.org/Submission/SWRL/`

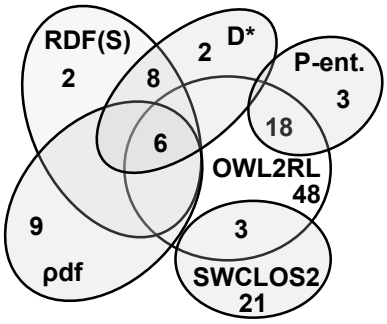

Fig. 1 Overlaps between the different sets of rules considered in the study. The numbers indicate the number of rules in each set or intersection.

can be observed. Beyond the high level view of Figure 1, there are cases where one rule of a regime (e.g. `rdfD1` in [W3C(2014b)]) overlaps two or more rules of another (`lg` and `rdf2-D` in [ter Horst(2005b)]). One regime may also implement two inverse variants of a rule (e.g. `prp-inv1` and `prp-inv2` in OWL 2 RL [W3C(2012)]), while another solution is to implement one of the rules complemented by an explicit inverse rule (`inv1` and `inv2` in [Komazec et al(2012)]). It is therefore observed that apart from D*, P and SWCLOS2, which are intentionally disjoint, there is no modularity between entailment regimes and they are not suitable for use in parallel.

3.1 Scope of the implemented entailment regimes

For purposes of this study entailments were generated exclusively through materialisation of triples by means of SPARQL Update rules. The tested sets of rule implementations and axiomatic triples as well as more detailed test results have been made available[15] together with sample results and instructions for repeating the tests described in this document. The exact scope of the implemented entailment regimes is:

- *RDF(S):* As listed and enumerated in the informative sections 8.1.1 (RDF) and 9.2.1 (RDFS) of [W3C(2014b)]. Even though the sections are informative, the rule identifiers in the list are dominantly used to refer to RDFS entailment rules in the referenced literature, prompting us to align with this approach for consistency and comparability. Therefore the rules for reflexivity of `subClassOf` and `subPropertyOf` have been excluded from the implementation of the RDFS regime, even though they are covered in the normative parts of the specification.
- *ρdf:* As specified in table 1 of [Muñoz et al(2009)].
- *D* and P:* As specified in tables 4[16] and 7 of [ter Horst(2005b)].
- *OWL 2 RL:* As specified in tables 4-9 of [W3C(2012)].
- *SWCLOS2:* As specified in tables 13.4 and 13.5 in the online manual referenced in Section 2.

3.2 Approaches to entailment rule implementation

The majority of entailment rules can be directly expressed in SPARQL as shown for `rdfs9` in Figure 2. The most common issue to address is the assumption of rule semantics that two variables with distinct labels do not assume equal values, whereas the following triples:

```
?x a owl:Class
?y a owl:Class
```

When used in a SPARQL query, would by default result in identical sets of solutions assigned to `?x` and `?y`. A typical outcome is that the output of a rule becomes a duplicate of one of the input conditions. The INSTANS platform does not re-insert an already existing triple, avoiding eternal loops in such cases, but they can also be avoided by filtering, as shown for `rdfs7` in Figure 2. Together these simple cases cover $> 84\%$ of all rules and unsatisfiability conditions in the tested sets.

```
# rdfs9 (cax-sco in OWL 2 RL, 3.(b) in rhodf)
INSERT { ?a a ?y }
WHERE { ?x rdfs:subClassOf ?y . ?a a ?x } ;
# rdfs7 (prp-spo1 in OWL 2 RL, 2.(b) in rhodf)
INSERT { ?s ?q ?o }
WHERE { ?s ?p ?o . ?p rdfs:subPropertyOf ?q
        FILTER (!sameTerm(?p,?q)) } ;
```

Fig. 2 SPARQL implementations of rdfs9 and rdfs7

The remaining 19 cases present interesting challenges, which are covered here by examples. OWL 2 RL uses lists in 12 rules. Even though it is possible to match the elements of a list to a variable by means of property paths[17], there are no means provided in SPARQL for indexed comparisons of the resulting elements. Therefore we have used a different approach, converting lists into indexed collections of elements as shown in Figure 3. As the assumption e.g. in LUBM (ref. Section 5) is that a list generated on the basis of the ontology is traversed multiple times by the instances in the data, the addition of the first list element is separated in our rules so that the list initialisation element is not deleted when the list is traversed. The OWL 2 RL unsatisfiability condition

[15] `https://github.com/aaltodsg/instans-reasoning`

[16] Rule `gl` not implemented.

[17] `http://www.w3.org/TR/sparql11-query/#propertypath-examples`

cax-adc, reproduced in Table 1, serves as an example of a rule which utilises a list with indexing, shown as SPARQL in Figure 4. This generic list approach was applicable to 9 rules.

```
PREFIX : <http://instans.org/>
# initiate a new list
INSERT { ?list ?rule_init [ :nexthead ?head ;
                            :listindex 1 ] }
WHERE { ?rule :listProperty ?lprop .
        ?list ?lprop ?head
        BIND (IRI(concat(str(?rule),"-init"))
              AS ?rule_init) } ;
# add the first list element, increment index
INSERT { ?list ?rule [ :nexthead ?nexthead ;
                       :listindex ?nextindex ; ] ;
           :element [ :value ?val ; :index ?index ] }
WHERE { ?rule :listProperty ?lprop .
        BIND (IRI(concat(str(?rule),"-init"))
              AS ?rule_init)
        ?list ?lprop ?head ;
              ?rule_init [ :nexthead ?thishead ;
                           :listindex ?index ] .
        ?thishead rdf:first ?val ;
                  rdf:rest ?nexthead .
        BIND (?index+1 as ?nextindex) } ;
# add new list element, increment index)
DELETE { ?list ?rule ?ref .
         ?ref :nexthead ?thishead ;
              :listindex ?index }
INSERT { ?list ?rule [ :nexthead ?nexthead ;
                       :listindex ?nextindex ; ] ;
           :element [ :value ?val ; :index ?index ] }
WHERE { ?rule :listProperty ?lprop .
        ?list ?lprop ?head ;
              ?rule [ :nexthead ?thishead ;
                      :listindex ?index ] .
        ?thishead rdf:first ?val ;
                  rdf:rest ?nexthead .
        FILTER (?nexthead != rdf:nil)
        BIND (?index+1 as ?nextindex) } ;
# add final list element, remove head and index
DELETE { ?list :nexthead ?thishead ;
               :listindex ?index }
INSERT { ?list :element [ :value ?val ;
                          :index ?index ] }
WHERE { ?rule :listProperty ?lprop .
        ?list ?lprop ?head ;
              ?rule [ :nexthead ?thishead ;
                      :listindex ?index ] .
        ?thishead rdf:first ?val ;
          rdf:rest ?nexthead .
        FILTER (?nexthead = rdf:nil) } ;
```

Fig. 3 SPARQL rules for list processing

Table 1 cax-adc from [W3C(2012)]

If	then
T(?x, rdf:type, owl:AllDisjointClasses) T(?x, owl:members, ?y) LIST[?y, $?c_1$, ..., $?c_n$] T(?z, rdf:type, $?c_i$) T(?z, rdf:type, $?c_j$)	false for each $1 \leq i < j \leq n$

The largest gap between rule and SPARQL semantics was found in prp-key from OWL 2 RL (Table 2). The intention is that **iff** ?x and ?y have properties matching all the

```
# declare
INSERT DATA { :caxadc :listProperty owl:members } ;

SELECT DISTINCT ?err ?x ?c1 ?c2
WHERE {
  ?x a owl:AllDisjointClasses ;
     owl:members ?y ;
     :element [ :value ?c1 ; :index ?i1 ] ;
     :element [ :value ?c2 ; :index ?i2 ] .
  ?z a ?c1,?c2 .
  FILTER ( ?i1 < ?i2 )
  BIND ("cax-adc false: AllDisjointClasses have
         common subject" as ?err) } ;
```

Fig. 4 SPARQL implementation of cax-adc utilising the list handler of Figure 3.

predicates of the list of keys and those properties have values ?z matching each other, ?x and ?y are declared sameAs. Direct conversion to a SPARQL query would produce solutions as soon as any common triples between the list of keys, ?x and ?y are found. Therefore we have used similar iterative list processing as the one shown in Figure 3 where predicates from the list of keys, ?x and ?y are matched step by step until the end of the list of keys is detected. The exact solution is left out of the paper in the interest of space, but can be found in our referenced github repository under *owl2rl-rules*. The complete statistical breakdown of the approaches used in implementing different rules and unsatisfiability conditions is shown in Table 3.

Table 2 prp-key from [W3C(2012)]

If	then
T(?c, owl:hasKey, ?u) LIST[?u, $?p_1$, ..., $?p_n$] T(?x, rdf:type, ?c) T(?x, $?p_1$, $?z_1$) ... T(?x, $?p_n$, $?z_n$) T(?y, rdf:type, ?c) T(?y, $?p_1$, $?z_1$) ... T(?y, $?p_n$, $?z_n$)	T(?x, owl:sameAs, ?y)

Table 3 Primary approaches used in implementing rules and unsatisfiability conditions.

Approach	Rules	Share
Simple (e.g. Figure 2 rdfs9)	82	67.8%
!sameTerm (e.g. Figure 2 rdfs7)	20	16.5%
Generic list (Figure 3, e.g. Figure 4)	9	7.4%
Custom list (e.g. Table 2)	3	2.5%
Special filtering	5	4.1%
Other (e.g. split to 3 queries)	2	1.7%

3.3 Treatment of blank nodes

Some rules (e.g. `rdfD1` in RDF and `lg` in D*) produce blank nodes. SPARQL entailment specification [Glimm and Ogbuji(2013)] states that "new blank nodes introduced in the saturation process are not to be returned in the solutions". As blank nodes only have local scope (typically a graph), a renaming operation is necessary to avoid errors in processing input from multiple sources [W3C(2014a)]. Also INSTANS maintains a graph-specific list and renames all incoming blank nodes using a local naming scheme. The same naming scheme is used for blank nodes generated by the active rules. Therefore it is not possible to syntactically distinguish between *a blank node introduced in the saturation process* and *a blank node loaded as input data*. In order to fully address the point from [Glimm and Ogbuji(2013)], a list of either *the blank nodes loaded as input* (to be included in solutions) or *the blank nodes materialised in the process* (to be excluded) would have to be maintained and queries would need to filter results accordingly. For the purposes of this paper the queries in both the conformance tests and the benchmark have been kept exactly as-is, and we have not implemented a global solution for blank node filtering. For those conformance tests, where unwanted blank nodes have been generated, the filtering has been performed in custom rules.

3.4 Event-based memory handling

As pointed out in [Pérez-Urbina et al(2012)], materialisation works best when both the ontology and the data are stable. Event processing, working on constantly changing infinite streams of data, would therefore seem like an especially bad match for materialisation. Generation of implicit data on an infinite stream will eventually fill the capacity of any computer or database.

Even though materialisation causes problems in an unlimited scenario, most stream processing tasks have limitations of scope also for the purpose of answering queries in finite time. In *data stream processing* queries are restricted to time windows [Barbieri et al(2010a)], enabling expiration of materialised triples together with the time windows. Also in *event processing* scenarios there is typically a maximum time, after which an old event can no longer impact the result. Sometimes all queries are event-specific, in which case materialisations may expire together with each event. When all the data with equal expiration time (e.g. a time window or a group of events with associated materialisations) is kept in a separate named graph, it can be expired and deleted without touching other graphs. To support removal of event information the separation of semi-static background knowledge (including e.g. all ontological data and materialisations based directly on the ontology) from the more dynamic event data is necessary, so that the background knowledge can be preserved.

Separating static and event data into separate graphs causes modifications to the entailment rules. For each triggering condition or materialised triple it needs to be known, whether said triple belongs to static or event data. An example of `rdfs2` implemented for LUBM is shown in Figure 5. Covering all possible combinations would explode the complexity of the system, but for many applications - including LUBM - the scope of each rule element is known from the context.

```
BASE <http://instans.org/>
INSERT { ?x a ?c }
WHERE { GRAPH <static> { ?p rdfs:domain ?c }
        ?x ?p ?y }
```

Fig. 5 Event-optimised implementation of `rdfs2` (`prp-dom` in OWL 2 RL, 4.(a) in ρdf).

To enable the switch of move operations between different graphs, the event stream needs to have markers, which can be matched by SPARQL. These markers can be individual triples or longer patterns, as long as they uniquely identify the necessary switching points. Markers are needed for switching between static and event data, as well as for recognising the borders between events.

The LUBM data consists of *universities*, which are further split into *departments*. No LUBM query requires matching between departments, allowing us to treat departments as tumbling time windows in a stream. They are expired and deleted, together with department-originated materialisations, each time a new department starts in the incoming data. LUBM does not have dedicated markers for the purpose, but we use triple pattern `?x a ub:Department` to switch from static to event input and triple pattern `?x a owl:Ontology`, which occurs in the beginning of every department, to switch between events. The general approach is illustrated in Figure 6, including the following steps:

1. All the SPARQL rules and queries (control rules [Figure 7], entailment rules, and the application queries [e.g. a LUBM query]) are loaded onto the platform. A `<control>` graph triple is set to `<static>` (Figure 7 `# Activegraph-init`).
2. Input to the main graph is handled as static data. If the data matches entailment rules, materialisations are written into the main graph.
3. Contents of the main graph, including the materialised triples, are continuously moved into the `<static>` graph (Figure 7 `# Move-static-input`).

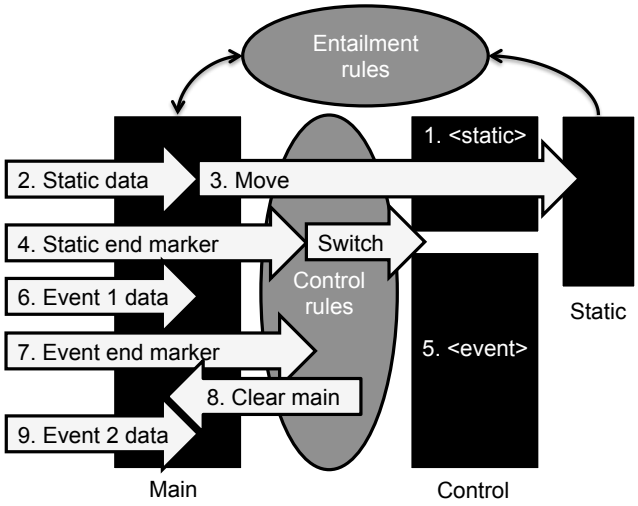

Fig. 6 Procedure for event-optimised memory handling.

4. The marker for switching to event input is received (`?x a ub:Department`).
5. A control rule switches a `<control>` graph triple from `<static>` to `<event>` (Figure 7 `# Change-to-event-mode`).
6. Data for the first event is received. Entailment rules use both the `<static>` and main graphs for input as specified (e.g. Figure 5). Results are generated by the application queries.
7. Event end marker is received (`?x a owl:Ontology`).
8. A control rule clears the main graph (Figure 7 `# Clean-previous-event`).
9. Data for the next event is received.

In current scenarios the complete background knowledge is typically available before any events, in which case there is no need to switch back to static data input. If there would be updates to the static data, a marker to designate switching from event input back to static data input and a corresponding control rule would be required. The efficiency of the *event-based memory handling* approach is investigated in Section 6.

4 Compliance Testing

SPARQL 1.1 Test Suite[18] has a separate section for tests on entailment. The list of entailment tests reported for implementations[19] includes 70 tests, but the 4 tests on *RIF* (Rule Interchange Format[20]) are not available in the actual test suite. The remaining 66 tests are split as follows:

[18] `http://www.w3.org/2009/sparql/docs/tests/`
[19] `http://www.w3.org/2009/sparql/implementations/`
[20] `https://www.w3.org/TR/rif-overview/`

```
BASE <http://instans.org/>
PREFIX : <http:instans.org/default#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/
                                    univ-bench.owl#>

# Activegraph-init
INSERT DATA {
  GRAPH <control> { :activegraph :address <static> } } ;

# Change-to-event-mode
DELETE { GRAPH <control> { :activegraph :address <static> } }
INSERT { GRAPH <control> { :activegraph :address <event> } }
WHERE { ?x a ub:Department
        GRAPH <control> { :activegraph :address <static> } } ;

# Clean-previous-event
DELETE { ?s ?p ?o }
WHERE {
  ?x a owl:Ontology
  GRAPH <control> { :activegraph :address <event> }
  ?s ?p ?o } ;

# Move-static-input
DELETE { ?s ?p ?o }
INSERT { GRAPH <static> { ?s ?p ?o } }
WHERE {
  GRAPH <control> { :activegraph :address <static> }
  ?s ?p ?o
  # Do not eat the change marker
  FILTER NOT EXISTS { ?x a ub:Department } } ;
```

Fig. 7 SPARQL input handler to separate static information from dynamic events (LUBM departments).

- 27 (40.9%) produce the reference result without applying any entailment rules.
- 3 (4.5%) have errors, e.g. different variables in the query and the sample answer.
- 36 (54.5%) need entailments to produce the reference results.

Out of 36, 16 tests were found to reproduce the reference solutions using one or more of the entailment regimes under study, as shown in Table 4. Test coverage for each regime is detailed in Table 5. OWL 2 RL shows the highest pass rate, but due to the large number of rules test set coverage is low and the number of rules in the regime per passed test is high. The best match with the test suite is demonstrated by ρdf, which passes a high number of tests with a small total number of rules. The most popular rule is `cax-sco` (Figure 2), which is stressed in up to 7 different tests. We performed further tests outside the test suite to verify the functionality of the rules falling outside test coverage.

Custom rules reproducing the reference results for 13 of the remaining tests were implemented. The reasons for customisation ranged from duplicate results (*sparqldl-13*) to requirements for materialisation of properties, which do not appear in any of the entailment regimes surveyed. As an example, a test denoted *simple4* expects to find pairs of elements using `unionOf`. The employed RDF construct is an ordered list, implying that instead of all two-element combinations a complete answer requires all ordered two-element permutations. To enable solution-specific filtering to avoid duplicating any part of the answer, the relatively complex

Table 4 SPARQL 1.1 entailment test mapping to regimes

Test	RDF(S)	ρdf	D*+P	OWL 2 RL
rdfs01	rdfs7	2.(b)	rdfs7x	prp-spo1
rdfs02	rdfs7	2.(b)	rdfs7x	prp-spo1
rdfs03	rdfs2 rdfs7	2.(b) 4.(a)	rdfs2 rdfs7x	prp-dom prp-spo1
rdfs04	rdfs9	3.(b)	rdfs9	cax-sco
rdfs05		3.(b) 7.(a)	rdfs9 rdfp5b rdfp9	cax-sco eq-ref3 scm-cls
rdfs06	rdfs2	4.(a)	rdfs2	prp-rng
rdfs07	rdfs3	4.(b)	rdfs3	prp-rng
rdfs09	rdfs9	3.(b)	rdfs9	cax-sco
rdfs11	rdfD2 rdfs6 rdfs7	2.(b) 6.(b)		
paper-sparqldl-Q1				eq-ref-1 scm-cls
paper-sparqldl-Q1-rdfs		3.(b) 7.(a)	rdfp5a rdfp9	
paper-sparqldl-Q4			rdfs9 rdfp5a rdfp9	cax-sco scm-cls
sparqldl-02		7.(b)	rdfp9 rdfp5a	cax-sco scm-cls
sparqldl-03		7.(b)	rdfp9 rdfp5a	cax-sco scm-cls
sparqldl-10			rdfp5a	eq-ref
parent3			rdfs9 rdfp12ab rdfp15 rule1 (SWCLS2)	cls-svf2 cax-sco scm-eqc1
parent9				scm-cls scm-sco scm-eqc1 scm-int
parent10				scm-sco scm-eqc1 scm-int

Table 5 Conformance test set coverages per entailment regime

Regime	Rules total	Rules tested	Test set coverage	Tests passed	Total per passed
RDF(S)	16	6	38%	8	2.0
ρdf	15	7	47%	12	1.3
D*	16	5	31%	7	2.3
D*+P	37	8	22%	13	2.8
OWL2RL	58	10	17%	16	3.6

pair of SPARQL rules shown in Figure 8 was constructed. For comparison a SPARQL query producing the reference test result directly from the test data is shown in Figure 9.

```
INSERT {
  ?x a [ a owl:Class ; owl:unionOf ( ?c0 ?c1 ) ] }
WHERE { ?c0 a owl:Class . ?c1 a owl:Class .
  ?x a ?c0 .
  FILTER (!sameTerm(?c0,?c1))
  FILTER (!isBlank(?c0))
  FILTER (!isBlank(?c1))
  FILTER NOT EXISTS { ?x a [ a owl:Class ;
                  owl:unionOf ( ?c0 ?c1 ) ] } } ;
INSERT {
  ?x a [ a owl:Class ; owl:unionOf ( ?c0 ?c1 ) ] }
WHERE { ?c0 a owl:Class . ?c1 a owl:Class .
  ?x a ?c1 .
  FILTER (!sameTerm(?c0,?c1))
  FILTER (!isBlank(?c0))
  FILTER (!isBlank(?c1))
  FILTER NOT EXISTS { ?x a [ a owl:Class ;
                  owl:unionOf ( ?c0 ?c1 ) ] } } ;
```

Fig. 8 SPARQL rules to pass test *simple4*.

```
SELECT DISTINCT ?x
WHERE { { ?x a :B } UNION { ?x a :C } } ;
```

Fig. 9 SPARQL query producing the same result as test *simple4*.

Utilising test-specific rulesets from both the referenced entailment regimes and the customised rules INSTANS passes 58 tests (89.4%). The main reasons for not passing the final 7 tests were the lack of support for the *nonNegativeInteger* type (4 tests) and *ASK-queries* (2 tests). After removal of missing and erroneous tests from the published test results the reported comparison pass rates for Jena, Pellet[21] and Stardog would be 55.6%, 88.9% and 38.1%, respectively[22]. It should also be noted that the test suite lists the same query twice with two alternative answers depending on the entailment regime, making a 100% pass rate using a single set of rules impossible.

5 Experimental setup

To compare the performance of complete entailment regimes with customised sets of rules and the performance of event-like processing with static memory handling, as well as the performance of INSTANS with other platforms

[21] `https://github.com/stardog-union/pellet`

[22] the results page does not include information on the program versions used to obtain the results

supporting reasoning, a benchmark was needed. We chose LUBM[23] [Guo et al(2004)] because it:

- has been successfully used in benchmarking for 10+ years
- has queries with and without reasoning requirements
- is not based on streaming data and can therefore be used also on non-streaming platforms
- features independent segments (departments), which can be used to simulate an event stream for the purpose of testing event-based memory handling

As LUBM requires entailment support beyond RDF(S) (Table 7) and is not based on a data stream, no data stream processors (C-SPARQL, Sparkwave or ETALIS) could be used for comparison. LUBM generator version *UBA 1.7* was used to generate data for 1, 5, 10 and 100 universities[24]. Both the data and the benchmark ontology were converted and packaged to single Turtle[25] files using the rdf2rdf[26] converter. Query syntax was aligned with SPARQL 1.1, otherwise the original queries were not touched.

INSTANS v. 0.3.0.1 was compiled using SteelBank Common Lisp (SBCL) v. 1.3.8 with a heap size of 32768M and executed from the command line. Stardog version 3.1.4 with default settings ("SL" reasoner) was used from the command line. A *reasoner-jena*[27] wrapper Scala[28] v. 2.11.7 application was created to support selecting the reasoner from command line and timing the execution, running Jena v. 3.0.0. Maximum heap size was set to (Xmx) 32G.

All experiments were executed on a MacBook Pro with a 2.7 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory running OS X Yosemite 10.10.5. All speed tests were executed four times, output was directed to `/dev/null` to mitigate impact of I/O, the first run was ignored as warm-up and the median value of the three remaining runs was recorded as the result. For INSTANS and Jena the timer was started before opening the data file and stopped at the end of iterating through the results. Stardog pre-reads files into a database in a separate step before processing. We used the "real" value of the unix-command "time" to measure the time used by Stardog. An average delay of running Stardog with no data[29] was subtracted as overhead and the time of loading the data into the database (Table 6) was added. It should, however, be noted that since the data is parsed into the database only once, the runtime experience for the Stardog end-user running individual queries is 3-63s faster than indicated in the results. Stardog suffered from some stability problems, with the server deterministically halting on the 3rd execution of Q6 through the batch. Re-starting the Stardog server for each trial stabilised the situation for batches of 5 and 10 universities[30].

Table 6 Number of triples and loading time measured by Stardog (U = University).

Set	Triples	Stardog load time [s]
1U	100,545	1.616
5U	624,534	3.086
10U	1,272,577	6.261
100U	13,405,383	63.636

6 Performance results

The LUBM web page provides reference query answers for one university. Even though LUBM is not an RL ontology, as it contains existentially quantified axioms, it was verified that all the platforms and qualifying sets of rules and reasoners perfectly reproduce the reference answers. A list of all 14 queries together with the minimum set of OWL 2 RL rules as well as the compliant reasoning frameworks for INSTANS and Jena are listed in Table 7. Q1-Q3 and Q14 need no reasoning. A maximum of three rules per query are required, and only eight rules in total are needed to pass all LUBM queries. The relative execution speed of each framework (reasoner or regime) was verified using five universities, the lists in Table 7 are ordered fastest first. Entries in parenthesis are theoretically compliant but failed to complete in practice using the complete regime.

Performance results for queries requiring no reasoning are shown in Figure 10. Q2 has high run-time memory requirements due to a large number of candidate solutions. For INSTANS Q2 event-based memory handling was required to complete a dataset of 5 or more universities, running at 4.2 ktriples/s. Jena Q2 performance was even lower and deteriorating for larger datasets at 0.67 (5U) and 0.36 (10U) ktriples/s. Comparing the 5U and 10U cases over the other queries we see that INSTANS performance is stable already at 5U, whereas Jena and Stardog continue to accelerate from 5U to 10U. The optimised reasoner of Stardog shows very minor impact on performance. Stardog with reasoner performs 34x-40x faster than event-based INSTANS without reasoning (10U case), indicating an upper limit for what could be achieved on INSTANS using a query rewriting reasoner (assuming no delay from the reasoner).

The impact of event-based memory-handling on INSTANS must be analysed over all test cases. Figure 11

[23] `http://swat.cse.lehigh.edu/projects/lubm/`
[24] seed 0, index 0
[25] `http://www.w3.org/TR/turtle/`
[26] `http://www.l3s.de/~minack/rdf2rdf/`
[27] `https://github.com/aaltodsg/reasoner-jena`
[28] `http://www.scala-lang.org/`
[29] ∼2.2s with reasoning, ∼1.5s without
[30] INSTANS and Jena also re-start completely for each execution.

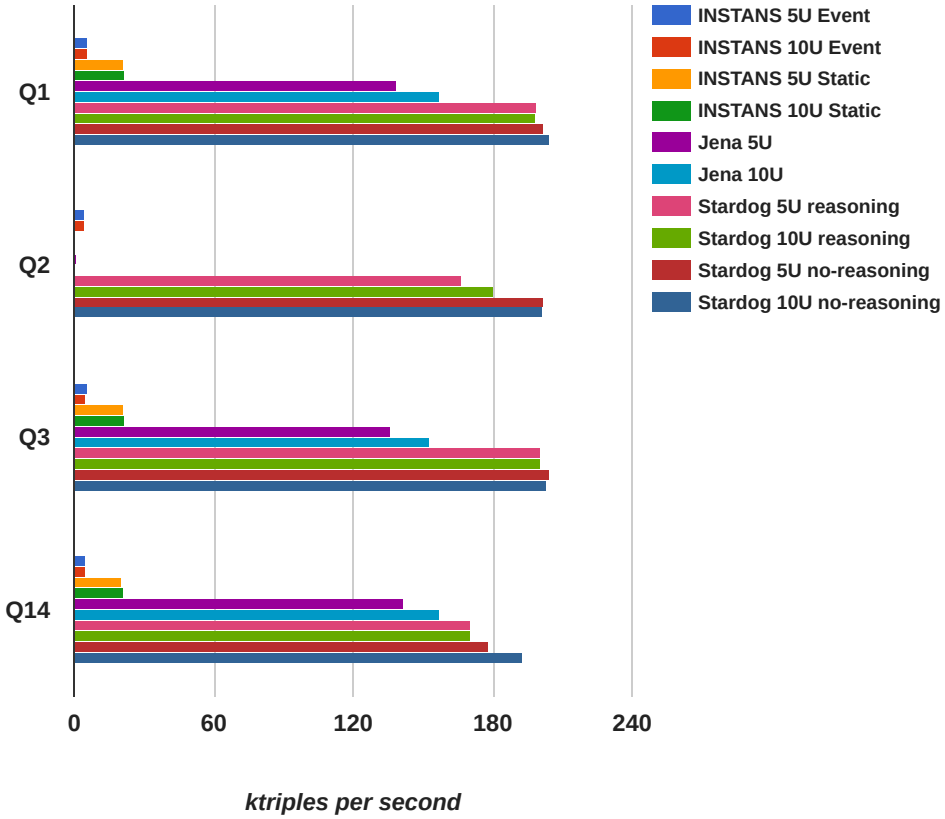

Fig. 10 Execution speed results for queries requiring no entailment (ktriple = 1,000 triples).

Table 7 LUBM queries, required rules using OWL 2 RL names, qualifying entailment regimes and Jena reasoners (fastest first, ()=did not complete in test).

Query	Rules	INSTANS	Jena
Q1-Q3	none		
Q4	cax-sco	RDFS ρdf D* (OWL2RL)	RDFSSimple RDFS OWLMicro OWLMini OWL
Q5	prp-dom prp-spo1 cax-sco	RDFS ρdf D* (OWL2RL)	RDFSSimple RDFS OWLMicro OWLMini OWL
Q6 - Q10	cls-int1 cax-sco scm-svf1	(OWL2RL)	OWL (Q7)
Q11	prp-trp	P-entail (OWL2RL)	OWLMicro OWLMini OWL
Q12	cls-int1 cax-sco cls-svf1	(OWL2RL)	OWL
Q13	prp-dom prp-spo1 prp-inv1	D*+P-ent (OWL2RL)	OWLMicro OWLMini OWL
Q14	none		

shows the INSTANS memory allocation sampled once per second for both event-based and static memory handling approaches using the OS X system utility `top` running Q9 (1U). The static approach exhausted heap memory and did not complete, whereas the event-based memory handling shows much better stability and faster execution. Without entailments (Figure 10) the static approach consistently performs ∼4.1x faster than event-based (10U). Also with limited entailment regime queries (Figure 12) static was 3.6x-5.5x faster (5U opt). In other queries (Figure 13 and Figure 14) the static approach collapses: Q7 and Q9 ran out of memory, Q12 did not complete in 44 hours (<3.9 triples/s). For the ones that did complete (Q6, Q8, Q10) the situation reversed and event-based was measured to be ∼24x faster in all three cases. This clearly shows how different aspects of memory pressure impact performance: in simpler cases the delay of erasing the main graph after each department (event-based approach) is significant and the static approach is faster, whereas in the more complex cases the large amount of in-memory bindings in the static case collapse performance and the event-based approach performs better.

Without entailments (Figure 10) Jena was measured ∼27x faster than INSTANS (5U Event) with the exception of Q2, where INSTANS was 6.2x faster than Jena. Mov-

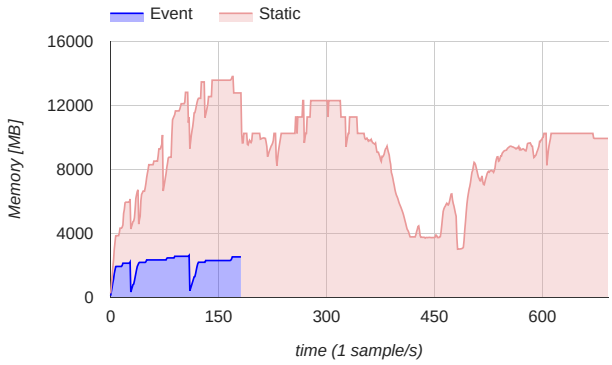

Fig. 11 INSTANS memory allocation for 1 university in Q9.

ing to queries with smaller entailment regimes (Figure 12) using the simplest and fastest regimes available on both INSTANS and Jena (Table 7) producing the correct results, Jena performed 9.1x-68.3x faster than INSTANS. Using the best measured performance (static case) with optimised rules INSTANS performance improves 1.3x-12x, leaving Jena 4.5x-7.4x faster and Stardog 9x-15x faster.

The situation changes dramatically for Jena with the queries requiring the most complete *OWL* reasoner. Q7 failed to complete on a single university, other queries took 1-27 hours. With 5U Q6 did not complete a single iteration in 59 hours (less than 3 triples/s). The complete OWL 2 RL on INSTANS (using the static approach) did not complete any query either. Using only the necessary rules and event-based processing, INSTANS successfully completed also all remaining queries (Figure 13, Figure 14 and Table 8). Stardog performed 18x-364x faster than INSTANS (10U Event). As the study focuses on the ease of customisation using SPARQL vs. pre-existing reasoner frameworks, not a head-to-head comparison of the platforms, no rule customisation on Jena or Stardog was performed.

To confirm stability, a batch of 100 universities was also tested on INSTANS and Stardog. The execution times of the best results achieved on each platform are shown in Table 8. All results on INSTANS are with optimised query sets; Q1, Q3, Q4, Q13 and Q14 with static memory handling and Q2, Q5-Q12 with event-based memory handling. Performance was in line with earlier results and the test confirmed that event-based memory handling remained stable over a dataset of 100 universities. Stardog demonstrated excellent performance in running the queries. The Stardog results are dominated by the loading time of the data to the database. Despite server restarts before each query, Q6 ("Unable to allocate 350.3M bytes, direct memory exhausted") and Q14 ("GC overhead limit exceeded") stopped with errors and Q10 repeatedly froze, leaving some gaps into the results.

Table 8 Fastest processing times [hh:mm:ss] for INSTANS and Stardog on 100 Universities LUBM.

Query	INSTANS	Stardog
Q1	00:10:02	00:01:03
Q2	04:45:05	00:01:04
Q3	00:10:15	00:01:03
Q4	00:12:00	00:01:04
Q5	01:28:53	00:01:03
Q6	01:13:00	
Q7	01:21:41	00:01:05
Q8	01:12:09	00:01:06
Q9	07:22:30	00:01:06
Q10	01:12:10	
Q11	00:50:08	00:01:03
Q12	01:32:15	00:01:03
Q13	00:18:40	00:01:03
Q14	00:10:37	

7 Conclusions

The feasibility of supporting reasoning with interconnected SPARQL Update rules has been assessed. Entailment regimes RDF(S), ρdf, D*, P, OWL 2 RL, rules from SWCLOS2 documentation as well as test-focused custom rules were implemented as a collection of 215 SPARQL queries and rules, which are hereby made available. Special treatment by interconnected SPARQL Update rules was required for 18 cases, described in this paper through examples and for every case in the complementary web resources. The collection is far more complete than seen in earlier documents of stream reasoning [Barbieri et al(2010a), Komazec et al(2012)]. These rule libraries are usable on any tool supporting networks of SPARQL Update rules.

The pass rate improvement[31] of each regime over the 66 entailment-tests of the *SPARQL 1.1 Test Suite* was found to be 24% OWL2 RL, 20% D*+P, 18% ρdf, 12% RDF(S) and 11% D*. The highest result for OWL 2 RL as the most complete regime was not as surprising as the good test compatibility of ρdf, which needed only 1.3 rules per passed test whereas OWL 2 RL used 3.6. Test coverages between 17-47% of the entailment regimes were observed, indicating that in order to properly test conformance, further development of the suite is necessary.

In the introduction the importance for the end-user to have full visibility and control over the entailment rules being used was stressed. This assumption was confirmed in multiple ways by the study:

- No more than four rules out of any regime were required for passing any single conformance test.
- Test-specific sets of rules were utilised to demonstrate a pass rate of 89.4%, higher than the reported result for any other platform at the time of writing.

[31] in addition to the 40.9% passable without entailments

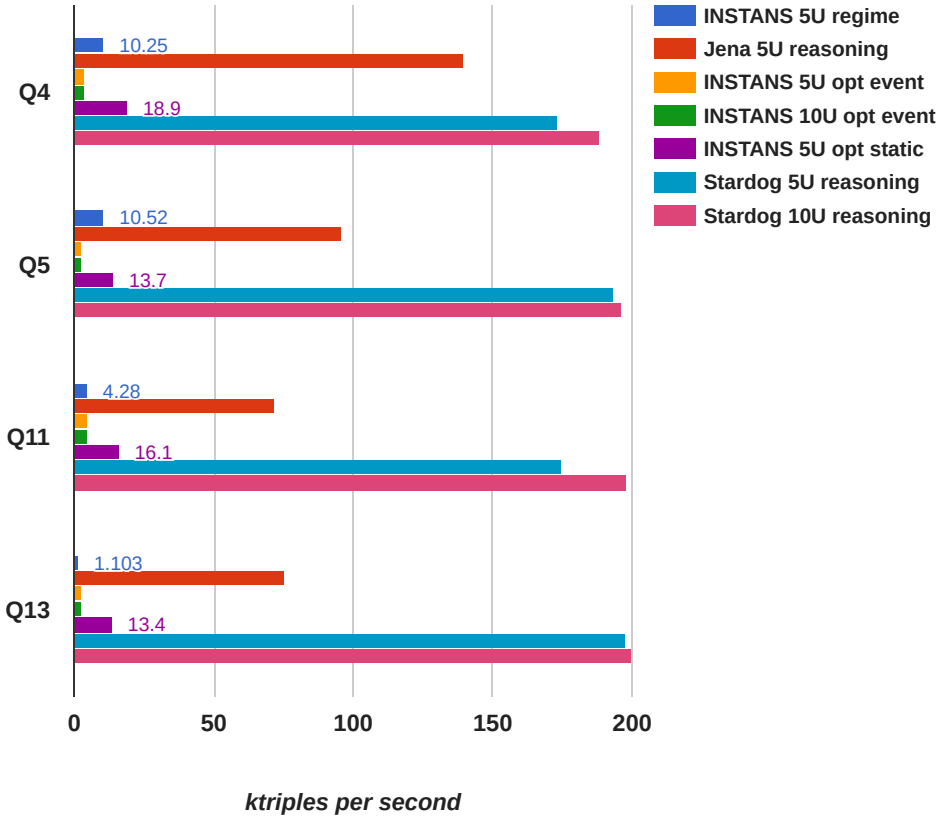

Fig. 12 Execution speed results for queries requiring small entailment regimes.

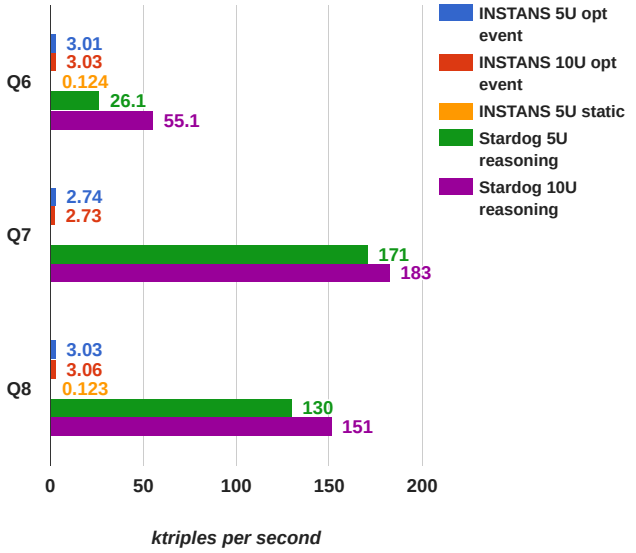

Fig. 13 Execution speed results for Q6-Q8.

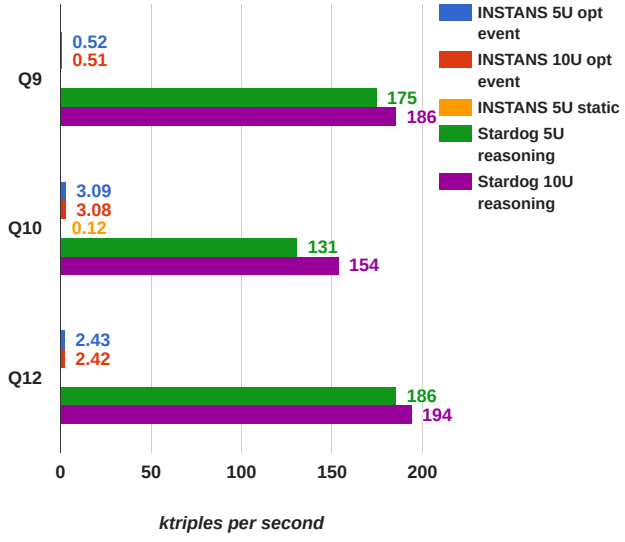

Fig. 14 Execution speed results for Q9, Q10 and Q12.

- Despite commonality between RDF(S), D* and ρdf, significant differences were discovered in the tests passable by the different regimes.
- For LUBM a maximum of three rules per query or a total set of eight OWL 2 RL rules are sufficient to pass.
- No tested pre-packaged reasoner completed five universities flawlessly on the LUBM queries requiring OWL entailment: INSTANS *OWL 2 RL* crashed the heap on all cases, Jena *OWL* did not complete any query in reasonable time and Stardog *SL* – while delivering impressive performance – required server re-starts between runs.
- Especially with materialisation-based reasoning the quantity and type of entailment rules have a critical impact on performance and stability.

The demonstrated solution leaves full control to the end-user. Entailment rules are written in the same language as the queries, so the end-user only needs to use one language. Over 84% of the examined rules and unsatisfiability conditions were shown to have a simple and straightforward conversion to SPARQL. A generic list processing approach covered 7% of the remaining cases. The main approaches were illustrated by examples. All rule implementations are available in the complementary repository. The rules are not integral to the platform and can be freely examined, swapped, edited and packaged together with the queries all the way to saving both rules and queries into the same SPARQL file. This ensures platform-independent versioning of rule frameworks and improves transparency and repeatability of experiments.

Materialisation as a method of reasoner implementation consumes capacity (whether the inferred triples are stored in memory or saved to a database). This does not match well with stream processing, where data is assumed to form an infinite stream. Despite that the presence of independent data segments such as time windows, events or university departments (LUBM) often enables construction of a case-specific solution, which separates the more static background knowledge (including the ontology and ontology-based materialisations) from the dynamic data (events and event-based materialisations) by using named graphs. A solution for this *event-based memory handling* on LUBM was demonstrated and tested to work on all queries up to a dataset of 100 universities. Memory consumption decreased and stabilised, observed best in the case of Q9, where a single university crashed 32GB of heap memory without the event-based solution, whereas 100 universities passed without problems with the solution activated.

Successful reproduction of LUBM reference answers with all queries both for our rule implementation on INSTANS and on two non-streaming SPARQL platforms was verified. Using only the necessary rules on INSTANS improved performance 1.3x-12.2x compared to the fastest-performing complete regime on each query. Stardog with an optimised query rewriting reasoner demonstrated clearly superior performance. Despite the lower execution speed, INSTANS with event-based memory handling and optimised sets of rules was the only tested platform, which successfully completed all LUBM queries up to 100 universities.

The optimisation techniques available for non-streaming platforms like Stardog and RDFox are different from continuously evaluating stream processors like INSTANS, as are the target application areas. The measured performance for INSTANS, 0.5 - 22.3 ktriples/s (over 100 LUBM universities), would be sufficient to support most examples (traffic monitoring, weather monitoring, manufacturing logistics, social media analysis etc.) used in stream processing scenarios. The usefulness of a stream processor is ultimately determined by the real-life use cases it can support.

As a future task a query rewriting reasoner should be tested over INSTANS. Even though INSTANS performance was not found to be on par with high-performance non-streaming platforms, the functionality of a pure SPARQL reasoner implementation in stream reasoning context was verified, leaving the door open for further performance improvements as required by future real-time streaming applications.

Acknowledgments

This work has been carried out in the TrafficSense project funded by Aalto University.

References

Anicic et al(2011). Anicic D, Fodor P, Rudolph S, Stojanovic N (2011) EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW '11 Proceedings of the 20th international conference on World wide web, ACM, Hyderabad, India, WWW '11, pp 635–644, DOI 10.1145/1963405.1963495

Barbieri et al(2010a). Barbieri DF, Braga D, Ceri S, Della Valle E, Grossniklaus M (2010a) Incremental reasoning on streams and rich background knowledge. In: Proceedings of the 7th Extended Semantic Web Conference, vol 6088 LNCS, pp 1–15, DOI 10.1007/978-3-642-13486-9_1

Barbieri et al(2010b). Barbieri DF, Braga D, Ceri S, Grossniklaus M (2010b) An execution environment for C-SPARQL queries. In: Proceedings of the 13th International Conference on Extending Database Technology - EDBT '10, Lausanne, Switzerland, p 441, DOI 10.1145/1739041.1739095

Barbieri et al(2010c). Barbieri DF, Braga D, Ceri S, Valle ED, Grossniklaus M (2010c) C-SPARQL: A Continuous query language for RDF data streams. International Journal of Semantic Computing 04:3, DOI 10.1142/S1793351X10000936

Bischof et al(2014). Bischof S, Krötzsch M, Polleres A, Rudolph S (2014) Schema-Agnostic Query Rewriting in SPARQL 1.1. In: ISWC 2014, vol 8796, pp 584–600, DOI 10.1007/978-3-319-11964-9_37

Forgy(1982). Forgy CL (1982) Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1):17–37, DOI 10.1016/0004-3702(82)90020-0

Glimm and Ogbuji(2013). Glimm B, Ogbuji C (2013) SPARQL 1.1 Entailment Regimes W3C Recommendation 21 March 2013. URL http://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/

Guo et al(2004). Guo Y, Pan Z, Heflin J (2004) An Evaluation of Knowledge Base Systems for Large OWL Datasets. In: ISWC 2004, vol 3, pp 274–288, DOI 10.1007/978-3-540-30475-3_20

ter Horst(2004). ter Horst HJ (2004) Extending the RDFS Entailment Lemma. In: ISWC 2004, pp 77–91, DOI 10.1007/978-3-540-30475-3_7

ter Horst(2005a). ter Horst HJ (2005a) Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity. In: Gil Y, Motta E, Benjamins VR, Musen MA (eds) ISWC 2005, Springer Berlin Heidelberg, pp 668–684, DOI 10.1007/11574620_48

ter Horst(2005b). ter Horst HJ (2005b) Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web 3(2-3):79–115, DOI 10.1016/j.websem.2005.06.001

Imprialou et al(2012). Imprialou M, Stoilos G, Grau BC (2012) Benchmarking Ontology-Based Query Rewriting Systems. In: Twenty-Sixth AAAI Conference on Artificial Intelligence, AAAI Press, pp 779–785

Komazec and Cerri(2011). Komazec S, Cerri D (2011) Towards Efficient Schema-Enhanced Pattern Matching over RDF Data Streams. In: Workshop on Ordering and Reasoning (ORDRING 2011), Springer, Bonn, Germany

Komazec et al(2012). Komazec S, Cerri D, Fensel D (2012) Sparkwave : Continuous Schema-Enhanced Pattern Matching over RDF Data Streams. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, ACM, pp 58–68, DOI 10.1145/2335484.2335491

Margara et al(2014). Margara A, Urbani J, Van Harmelen F, Bal H (2014) Streaming the Web: Reasoning over dynamic data. Journal of Web Semantics 25:24–44, DOI 10.1016/j.websem.2014.02.001

Muñoz et al(2007). Muñoz S, Pérez J, Gutiérrez C (2007) Minimal Deductive Systems for RDF. In: The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, vol 4519, pp 53–67, DOI 10.1007/978-3-540-72667-8_6

Muñoz et al(2009). Muñoz S, Pérez J, Gutierrez C (2009) Simple and Efficient Minimal RDFS? Web Semantics: Science, Services and Agents on the World Wide Web 7(3):220–234, DOI 10.1016/j.websem.2009.07.003

Nenov et al(2015). Nenov Y, Piro R, Motik B, Horrocks I, Wu Z, Banerjee J (2015) RDFox : A Highly-Scalable RDF Store. In: The Semantic Web - ISWC 2015 - Part 2, Springer International Publishing, pp 3–20, DOI 10.1007/978-3-319-25010-6{_}1

Pérez-Urbina et al(2012). Pérez-Urbina H, Rodríguez-Díaz E, Grove M, Konstantinidis G, Sirin E (2012) Evaluation of query rewriting approaches for OWL 2. In: Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems - SSWS+HPCSW 2012, CEUR Workshop Proceedings, vol 943, pp 32–44, DOI 10.1.1.416.8559

Polleres et al(2013). Polleres A, Hogan A, Delbru R, Umbrich J (2013) RDFS and OWL reasoning for linked data, vol 8067 LNAI. Springer Berlin Heidelberg, DOI 10.1007/978-3-642-39784-4{_}2

Rinne and Nuutila(2014). Rinne M, Nuutila E (2014) Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL. In: Meersman R, Panetto H, Dillon T, Missikoff M, Liu L, Pastor O, Cuzzocrea A, Sellis T (eds) On the Move to Meaningful Internet Systems: OTM 2014 Conferences, Springer Berlin Heidelberg, pp 682–699, DOI 10.1007/978-3-662-45563-0_42

Singh and Karwayun(2010). Singh S, Karwayun R (2010) A Comparative Study of Inference Engines. In: 2010 Seventh International Conference on Information Technology: New Generations, IEEE, pp 53–57, DOI 10.1109/ITNG.2010.198

Valle et al(2013). Valle ED, Schlobach S, Krötzsch M, Bozzon A, Ceri S, Horrocks I (2013) Order matters! Harnessing a world of orderings for reasoning over massive data. Semantic Web Journal 4(2):219–231, DOI 10.3233/SW-2012-0085

Volz et al(2005). Volz R, Staab S, Motik B (2005) Incrementally maintaining materializations of ontologies stored in logic databases. Journal on Data Semantics 2:1–34, DOI 10.1007/978-3-540-30567-5_1

W3C(2012). W3C (2012) OWL 2 Web Ontology Language Profiles (Second Edition) W3C Recommendation 11 December 2012. URL http://www.w3.org/TR/owl2-profiles/

W3C(2013a). W3C (2013a) SPARQL 1.1 Query Language W3C Recommendation 21.3.2013. URL http://www.w3.org/TR/sparql11-query/

W3C(2013b). W3C (2013b) SPARQL 1.1 Update W3C Recommendation 21 March 2013. URL http://www.w3.org/TR/sparql11-update/

W3C(2014a). W3C (2014a) RDF 1.1 Concepts and Abstract Syntax W3C Recommendation 25 February 2014. URL http://www.w3.org/TR/rdf11-concepts/

W3C(2014b). W3C (2014b) RDF 1.1 Semantics W3C Recommendation 25 February 2014. URL http://www.w3.org/TR/rdf11-mt