Web Reasoning Using Fact Tagging

Mehdi Terdjimi
Univ Lyon, LIRIS - Université Lyon 1 - CNRS UMR5205
Villeurbanne, France
mehdi.terdjimi@liris.cnrs.fr

Lionel Médini
Univ Lyon, LIRIS - Université Lyon 1 - CNRS UMR5205
Villeurbanne, France
lionel.medini@liris.cnrs.fr

Michael Mrissa
Univ Pau & Pays Adour, LIUPPA, EA3000
Pau, France
michael.mrissa@univ-pau.fr

ABSTRACT

Today's Web applications tend to reason about cyclic data (i.e. facts that re-occur periodically) on the client side. Although they can benefit from efficient incremental maintenance algorithms capable of handling frequent data updates, existing rule-based algorithms cause successive re-derivations of previously inferred information. In this paper, we propose an incremental maintenance approach for rule-based reasoning that prevents successive re-computations of fact derivations. We tag (i.e. annotate) facts to keep trace of their provenance and validity. We compare our solution with the DRed-based incremental reasoning algorithm and show that it significantly outperforms this algorithm for fact updates in re-occurring situations, to the cost of tagging facts at their first insertion. Our experiments show that this cost can be recovered within a small number of cycles of deletions and reinsertions of explicit facts. We discuss the utility and limitations of our approach on Web clients and provide implementation packages of this reasoner that can be directly integrated in Web applications, on both server and client sides.

CCS CONCEPTS

• **Theory of computation** → **Semantics and reasoning**; • **Computing methodologies** → **Causal reasoning and diagnostics**; • **Information systems** → *Web applications*; *Web Ontology Language (OWL)*;

KEYWORDS

rule-based reasoning; Web reasoning; incremental fact maintenance; fact tagging

ACM Reference Format:
Mehdi Terdjimi, Lionel Médini, and Michael Mrissa. 2018. Web Reasoning Using Fact Tagging. In *The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3184558.3191615

This paper is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution. In case of republication, reuse, etc., the following attribution should be used: "Published in WWW2018 Proceedings © 2018 International World Wide Web Conference Committee, published under Creative Commons CC BY 4.0 License."
WWW'18 Companion, April 23–27, 2018, Lyon, France
© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.
ACM ISBN 978-1-4503-5640-4/18/04..
https://doi.org/10.1145/3184558.3191615

1 INTRODUCTION

Everyday Web applications must dynamically handle various types of contents generated by users or client sensors. Semantic technologies could improve these applications, but are currently under-exploited. One reason is that full-fledged semantic stacks are perceived as costly, unreliable server-sided architectures, in opposition with current (i.e. modular and client-side) Web design practices [19]. We have addressed this problem in [18], by proposing HyLAR, a reasoner that can be both used on the server and client sides. Our goal is to allow using reasoning for tasks currently located on Web application clients, that satisfy several conditions. We focus on datasets of relatively small size (< 50k triples) and target Web applications based on stable data models (TBoxes) and more varying model instances (ABoxes). State-of-the-art in semantic reasoning research work [13] aims at improving reasoning through maintenance algorithms such as incremental reasoning (IR). However, when the updated data is cyclic (i.e. facts that re-occur periodically), applications should not only rely on IR to optimize reasoning, as they are regularly exposed to overheads caused by re-deriving implicit facts that have been already derived in the past.

In this paper, we propose a maintenance approach inspired by IR that prevents successive re-derivations by tagging facts with respect to their provenance and validity. Our solution includes the following contributions:

- **Faster deletions using validity tagging.** We provide validity tagging for explicit facts and do not process overdeletion tasks. Instead, explicit facts are tagged as invalid at deletion time and as valid at reinsertion time.
- **Faster re-insertions using provenance tagging.** We track the provenance of all implicit facts (i.e. all possible derivations), which avoids having to re-evaluate them if they are reinserted in the knowledge base.
- **Reasoning on the Web.** We provide a rule-based reasoner that currently supports a subset of OWL 2 RL rules, usable on both JavaScript-enabled servers and Web browsers.

This paper is structured as follows. Section 2 formalizes and highlights the re-derivation overhead problem, in a scenario involving a mobile Web application. Section 3 presents our contribution with three algorithms: implicit fact tagging, tag-based KB update and fact selection filtering. Section 4 describes our prototype and enumerates the entailment rules it is currently capable to handle. Section 5 evaluates our solution by comparing it with IR and discusses the results with respect to different application settings. Section 6 overviews

related work on reasoning profiles and optimizations. Section 7 concludes and draws perspectives of our work.

2 PROBLEM STATEMENT

Web applications can be subject to frequent updates. Possibly re-occurring data can be re-inserted or re-deleted, which can cause significant computational overheads. We illustrate this issue with the scenario of a mobile Web application connected to a smart house: Julia uses this application on her smartphone to automatically regulate her house temperature when she approaches her house. The application locates her mobile phone either using its GPS sensor or by recognizing the network it is connected to. She will be considered close to her house either if her cell phone GPS coordinates correspond to her house neighborhood or if she connects the phone to the house local network. This activates temperature regulation and deactivates it otherwise. Julia's proximity from her house is the re-occurring data: the application infers or not this information as she moves back and forth with her cell phone, as she switches on and off the GPS sensor, or as she connects and disconnects her phone from the house network.

We use the following formalization, from Motik et al. [13]: a fact F can be explicit (*i.e.* provided at startup or update), implicit (*i.e.* derived as a rule consequence), or both implicit and explicit (*i.e.* explicitly stated and derived). A rule r has an antecedent, conjunction of facts F_i, $i \in \mathbb{N}$) and an implied consequence (a single fact I); when it applies, the consequence is derived as an implicit fact: r :- $F1 \wedge F2 \wedge ... \wedge Fx \rightarrow I$.

Application ontology. Julia's application in our scenario uses the following fixed ontology (Classes and Properties) and entailment rules.

```
EO1 = :PhysicalAgent rdf:type owl:Class .
EO2 = :User rdfs:subClassOf :PhysicalAgent .
EO3 = :SmartDevice rdfs:subClassOf :PhysicalAgent .
EO4 = :SmartPhone rdfs:subClassOf :SmartDevice .
EO5 = :SmartHome rdfs:subClassOf :SmartDevice .
EO6 = :Location rdf:type owl:Class .
EO7 = :TemperatureStatus rdf:type owl:Class .
```

Listing 1: Classes

```
EO8 = :hasLocation rdf:type owl:ObjectProperty
      .
EO9 = :hasLocation rdfs:domain :PhysicalAgent .
EO10 = :hasLocation rdfs:range :Location .
EO11 = :hasLocationCloseTo rdf:type
       owl:ObjectProperty .
EO12 = :hasLocationCloseTo rdf:type
       owl:TransitiveProperty .
EO13 = :hasLocationCloseTo rdfs:domain
       :PhysicalAgent .
EO14 = :hasLocationCloseTo rdfs:range
       :PhysicalAgent .
EO15 = :hasTemperatureRegulation rdf:type
       owl:ObjectProperty .
EO16 = :hasTemperatureRegulation rdfs:domain
       :SmartHome .
EO17 = :hasTemperatureRegulation rdfs:range
       :TemperatureStatus .
```

Listing 2: Properties

```
Transitivity : (?p rdf:type owl:TransitiveProperty)
  ∧ (?i1 ?p ?i2) ∧ (?i2 ?p ?i3) → (?i1 ?p ?i3)
Subsumption: (?c1 rdfs:subClassOf ?c2)
  ∧ (?s rdf:type ?c1) → (?s rdf:type ?c2)
```

Listing 3: Entailment rules

Application instances and rules. Below are the initial explicit and implicit facts inferred via the Business Rules (Listing 4), which drive the application behavior. The set of initial explicit facts declares Julia, her cell phone, her house and the instance that activates temperature regulation in the KB, and assumes that Julia always carries her cell phone with her. The application can reason about their locations via $r1$ (as they are inferred as physical agents), and can switch on the regulation via $r2$.

```
r1: (?agent :hasLocation
            :JuliasHouseNeighborhoodLocation
            )
→ (?agent :hasLocationCloseTo :JuliasHouse)
r2: (:Julia :hasLocationCloseTo :JuliasHouse)
→ (:JuliasHouse :hasTemperatureRegulation
            :Activated)
```

Listing 4: Business rules

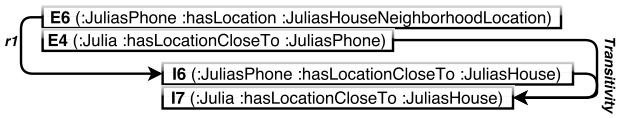
```
E1 = :Julia rdf:type :User .
E2 = :JuliasPhone rdf:type :SmartPhone .
E3 = :JuliasHouse rdf:type :SmartHome .
E4 = :Julia :hasLocationCloseTo :JuliasPhone .
E5 = :Activated rdf:type :TemperatureStatus .
```

Listing 5: Initial explicit facts

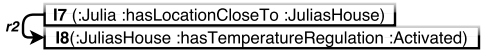
```
I1 = :JuliasPhone rdf:type :SmartDevice .
I2 = :JuliasHouse rdf:type :SmartDevice .
I3 = :Julia rdf:type :PhysicalAgent .
I4 = :JuliasPhone rdf:type :PhysicalAgent .
I5 = :JuliasHouse rdf:type :PhysicalAgent .
```

Listing 6: Initial implicit facts (inferred instances via subsumption)

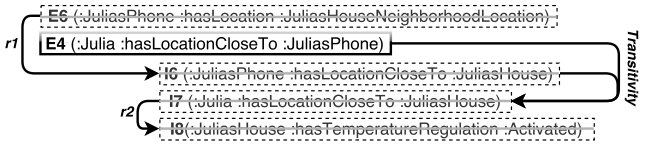
We consider the following 3-steps scenario.
(1) Julia approaches her neighborhood with her cell phone. The application analyzes the phone GPS coordinates and adds the explicit fact E6. This allows the reasoner to infer I6 via $r1$ and I7 via *Transitivity*.



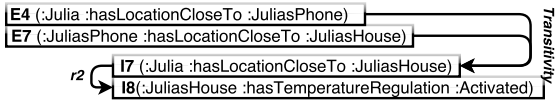
The application then enables temperature regulation as I7 triggers I8 via $r2$.



(2) Julia enters her house and cuts off the GPS to save energy. The phone position becomes unknown. The application removes E6, which also triggers the removal of I6, I7, I8, and disables temperature regulation.



(3) Julia connects her phone to the house local network. The application inserts E7, causing I7 and I8 to be re-derived respectively via $Transitivity$ and $r2$.



Step 3 highlights the re-evaluation overhead caused by over-deletion in the IR algorithm: the deletion and reinsertion of explicit facts leads to the re-derivation of two implicit facts that have already been derived at first insertion.

3 TAG-BASED INCREMENTAL MAINTENANCE

To avoid recurrent re-derivations, we propose to keep the origin of previously obtained inferences so that when already known facts re-occur, the reasoner can quickly retrieve their consequences. To do so, it must *keep track of all facts*, including deleted ones, and be able to *assess their validity*: explicit facts are tagged as valid/invalid, and implicit fact validity is retrieved using those of the explicit facts they have been derived from. When the reasoner receives an INSERT query, it only runs its inference algorithm on the explicit facts that have not been inserted before and simply validates the others. Processing DELETE queries only consists in invalidating the corresponding facts instead of removing them from the KB (as done in IR). At SELECT queries, the reasoner queries the knowledge base and filters the resulting facts according to their validity.

The speed of this process relies on the principle of storing explicit fact validity in memory and obtaining implicit fact validity from simple logic operations on these values: an implicit fact can originate from the disjunction of several sets of facts (explicit or implicit) that match the antecedent pattern of a same rule or from multiple rules, and rule antecedents are defined as conjunctions.

Finally, we introduce a *fact forgetting* mechanism to avoid KB inflation: each fact is tagged with the timestamp of its latest validity update, so that the oldest invalid facts are asynchronously removed when the KB size reaches a threshold.

In our scenario, when Julia switches the phone GPS off, the application "loses" its location and asks the reasoner to remove E6. But the reasoner only invalidates this fact. Then, the application sends a SELECT query on I8. The reasoner performs a simple logical operation (explained below) on I8 causes (E4, E6) that assesses that I8 is invalid, as E6 is invalid. It then does not return I8. When the phone connects to the house network, the application creates E7. The reasoner attaches it as alternative derivation of I7. At the next SELECT query, it deduces that I8 is valid as I7 is valid, and sends it back to the application. The next subsections detail the main elements of our Tag-Based (TB) maintenance approach: validity assessement, fact tagging, reasoning process and selection tasks.

3.1 Fact validity

Let F_e and F_i be respectively the sets of explicit and implicit facts in the KB. We propose to keep all facts (explicit and implicit) in the KB until the reasoning process is stopped or the fact forgetting mechanism triggered, and to assess their validity instead of removing them at DELETE queries. To do so, we tag explicit facts with a *valid* boolean indicator: $f_e.valid \in \mathbb{B}, f_e \in F_e$, which is set to *true* on insertion and *false* on deletion. We tag implicit facts with a *derivedFrom* indicator that represents the minimal set of disjoint *causes* of an implicit fact. We define a cause C as a set of explicit facts that must all be valid to validate an implicit fact[1]: $C = \{f_e \mid f_e \in F_e\}$. Hence, $\forall f_i \in F_i, f_i.derivedFrom = \{C_i\}, i \in \mathbb{N}/\forall x, y, 0 \leq x < y \leq i, C_x \nsubseteq C_y, C_y \nsubseteq C_x$.

We provide an $isValid()$ function that checks the validity of an implicit fact using its *derivedFrom* tag. It evaluates the disjunction between the tag elements and for each element, the conjunction between the *valid* tags of the explicit facts referenced in this element: $isValid(f_i) = \vee_{C_i}\{\wedge_{f_{ej}}\{f_{ej}.valid\}\}, C_i \in f_i.derivedFrom, f_{ej} \in C_i$. Implicit fact validity in our temperature regulation scenario is assessed as follows:

```
I6.derivedFrom = {{E6}}
isValid(I6) = E6.valid
I7.derivedFrom = {{E4,E6,EO12},{E4,E7,EO12}}
isValid(I7) = E4.valid ∧ EO12.valid ∧
              (E6.valid ∨ E7.valid)
I8.derivedFrom = I7.derivedFrom
isValid(I8) = isValid(I7)
```

Listing 7: Tagged implicit facts and corresponding validity assessment

3.2 Implicit fact tagging

Each time an implicit fact is derived, Algorithm 1 is applied to set its *derivedFrom* tag. Let F_e (resp. F_i) be the sets of explicit (resp. implicit) facts a newly inferred fact f have been derived from. In the general case, the algorithm builds the set *resolvedExplicitCauses* of resolved explicit causes by replacing implicit facts with their explicit causes[2] and

[1]To avoid recursion while assessing implicit fact validity, algorithm 2 (see below) only stores explicit facts in causes.

[2]These implicit facts have been inferred from prior evaluation loops; hence their derivedFrom tag is already set and stricly composed of explicit facts.

Algorithm 1 Implicit fact tagging

Require: A newly inferred implicit fact f and the sets F_e (resp. F_i) of explicit (resp. implicit) facts it has been derived from.
Ensure: f carries a *derivedFrom* tag composed of its explicit causes only.

```
1: if F_i = ∅ then
2:     f.derivedFrom ← {F_e}
3:     return f
4: end if
5: resolvedExplicitCauses ← F_i.first().derivedFrom
6: for all implicitFact ∈ (F_i \ F_i.first()) do
7:     tmp ← ∅
8:     for all resolved ∈ resolvedExplicitCauses do
9:         for all expDerivation ∈ implicitFact.derivedFrom do
10:            tmp ← tmp ∪ {resolved ∪ expDerivation}
11:        end for
12:    end for
13:    resolvedExplicitCauses ← tmp
14: end for
15: if F_e = ∅ then
16:    f.derivedFrom ← resolvedExplicitCauses
17:    return f
18: end if
19: explicitCauses ← ∅
20: for all resolved ∈ resolvedExplicitCauses do
21:    explicitCauses ← explicitCauses ∪ {resolved ∪ F_e}
22: end for
23: f.derivedFrom ← explicitCauses
24: return f
```

deduplicating these causes (lines 4-10). It then builds the set *explicitCauses* of explicit causes by distributing the initial set of explicit facts F_e into *resolvedExplicitCauses* (lines 14-16). It finally sets *explicitCauses* as *derivedFrom* tag of f – now tagged with a set of disjoint explicit causes – and terminates (lines 17-18).

Two optimizations allow avoiding unnecessary loops: (i) if no implicit fact is present (*i.e.* F_i is empty), the algorithm sets f.*derivedFrom* to F_e and terminates at line 3; (ii) if no explicit fact is present (*i.e.* F_e is empty), the algorithm sets f.*derivedFrom* to *resolvedExplicitCauses* and terminates at line 13.

3.3 Enabling tagging in reasoning

The KB update algorithm (Algorithm 2) performs the reasoning process while answering INSERT and DELETE queries. Let R be the set of rules and F_e^+ and F_e^- the sets of explicit facts to be respectively added and removed (from the query). It first invalidates the explicit facts to be deleted, and validates those to be inserted (lines 2-6), so that F_e^+ only contains new facts to be evaluated at line 7. Hence, for all deletions and re-insertions, our approach allows to skip the whole evaluation loop (lines 9-13).

Algorithm 2 Tag-based KB update

Require: Rule set R, explicit facts F_e, implicit facts F_i, added explicit facts F_e^+, removed explicit facts F_e^-
Ensure: The KB updates correspond to the changes caused by F_e^+ and F_e^- wrt. R.

```
1: F_i^+ ← ∅
2: for all fact ∈ F_e do
3:     if fact ∈ F_e^- then fact.valid ← false
4:     else if fact ∈ F_e^+ then
5:         fact.valid ← true
6:         F_e^+ ← F_e^+ \ {fact}
7:     end if
8: end for
9: if F_e^+ ≠ ∅ then
10:    F_e ← F_e ∪ F_e^+
11:    loop
12:        F_i ← F_i ∪ F_i^+
13:        R_kb ← restrictRuleSet(R, F_e ∪ F_i)
14:        F_i^+ ← evaluateRuleSet(R_kb, F_e ∪ F_i)
15:        F_i^+ ← combine(F_i, F_i^+)
16:        if F_i^+ ⊂ F_i then break
17:        end if
18:    end loop
19: end if
20: return F_e ∪ F_i
```

For the remaining facts in F_e^+, the evaluation loop works very similarly to IR [13]: the reasoner restricts R to the set R_{kb} of rules that match at least one cause in the updated KB ($F_e \cup F_i$) in *restrictRuleSet*() (line 11), evaluates R_{kb} over $F_e \cup F_i$ (*evaluateRuleSet*(), line 12) and loops as long as new implicit facts are inferred. TB reasoning requires two additional steps: (i) at each iteration, the *combine*() function deduplicates identical facts by concatenating their causes and removes unnecessary causes[3] (line 13), and (ii) when new implicit facts have been inferred (*i.e.* in the innermost loop of the *evaluateRuleSet*() function), it calls Algorithm 1 to set the fact causes in their *derivedFrom* tags (line 12). After the evaluation loop, the algorithm terminates and returns $F_e \cup F_i$, that reflects the KB changes, namely the updates in F_e^+ and F_e^- and the *valid* and *derivedFrom* tags of facts.

3.4 Fact-filtering

The fact-filtering algorithm (Algorithm 3) is applied after SELECT queries to filter out valid facts. As these queries are time-critical for the application and this step represents an overhead compared to other approaches, this algorithm must be kept fast. Let F be a query result set of facts. The algorithm performs a single loop over F to construct – and return – the set of valid facts V of F: $V = \{f_e \in F \cap F_e / F_e.valid = true\} \cup \{f_i \in F \cap F_i / isValid(F_e) = true\}$[4].

[3] For instance, if f_i can be caused by both $f_{e1} \wedge f_{e2}$ and $f_{e1} \wedge f_{e2} \wedge f_{e3}$, only the former conjunction is stored as a cause.
[4] For the sake of understandability, the algorithm comprises an $hasTag()$ function to filter explicit from implicit facts. This function is not implemented in practice.

Algorithm 3 Tag-based KB filtering

Require: F a set contaning explicit and implicit facts from a SELECT query answer.
Ensure: The returned set is composed of valid facts.
1: $V \leftarrow \emptyset$
2: **for all** $f \in F$ **do**
3: **if** $((hasTag(f, valid)$ **and** $f.valid)$ **or** $(isValid(f))$ **then** $V \leftarrow V \cup \{f\}$
4: **end if**
5: **end for**
6: **return** V

4 IMPLEMENTATION

Our approach targets a particular working context: OWL reasoning embedded in Web applications. This differs from traditional setups where regular OWL (e.g. Pellet[5]) or rule-based (e.g. RDFox[6] or CHR[7]) reasoners can be found. Hence, we implemented our approach in HyLAR[8] [17], a rule-based reasoner that includes both the IR algorithm from [13] and our tag-based algorithms: update (Algorithms 1 and 2) and selection (Algorithm 3) from Section 3. It currently processes the following subset of OWL 2 RL rules [12] (section 4.3):

R_{sub} = {scm-sco, cax-sco, scm-spo, prp-spo1} ($Subsumption$)
$R_{trans-inv}$ = {prp-trp, prp-inv1, prp-inv2} ($Transitivity/Inverse$)
R_{equiv} = {cax-eqc1, cax-eqc2, prp-eqp1, prp-eqp2} ($Equivalence$)
R_{equal} = {eq-rep-s, eq-rep-p, eq-rep-o, eq-trans} ($SameAs$)
$R_{all} = R_{sub} \cup R_{trans-inv} \cup R_{equiv} \cup R_{equal}$

Listing 8: HyLAR's sets of rules

We restrained to these rules as they seemed to us of reasonable complexity (regarding the fact that applications are supposed to perform reasoning tasks on diverse clients) whilst corresponding to the needs of "average" Web applications (i.e. manipulating typed objects – instances – and inheriting their properties from classes). However, this set of rules can be extended: the reasoner similarly processes entailment and business rules, application developers can add entailment rules among business logics ones, without modifying the reasoner package itself. HyLAR also supports consistency checking in both IR and TB algorithms. Entailment rules involving disjointness, complementness or type checking can be provided as built-in rules using the "false" fact as consequence. HyLAR handles this fact as any other and uses TB maintenance approach to efficiently perform rule-based consistency checking. HyLAR runs on both server (Node.js[9] 5.1.1) and client (Browserify[10]) sides, which allows its integration into frameworks designed to optimize reasoning process location, such as [18]. HyLAR benefits from JavaScript's asynchronous patterns (promises, callbacks), Web workers (on browsers) and event emitters (on Node.js), allowing for background task processing. It is composed of the modules depicted in Figure 1.

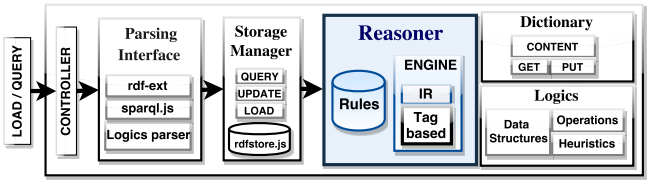


Figure 1: HyLAR global architecture.

- **Controller:** handles ontology loading (parsing and classification) and querying requests: inferences on INSERT and DELETE queries (using IR or tag-based algorithms) and filtered SELECT queries for tag-based reasoning.
- **Parsing Interface:** integrates rdf-ext's RDF/XML parser and SPARQL.js[11] library, and is able to convert triples (as described in RDF Interfaces 1.0[12]) into turtle (for direct triplestore insertion/deletion) and facts (for reasoning).
- **Storage Manager:** based on rdfstore.js[13] [7], this module handles ontology loading, updates and queries.
- **Reasoner:** holds and processes rules using a pattern matching mechanism; its engine both includes an incremental and a tag-based algorithms.
- **Dictionary:** indexes all triples registered in the store and their representations as facts in the KB; this accelerates validity checking at selection time.
- **Logics:** contains first-order logic operations: fact instantiation, fact tagging, fact merging ($combine()$), and rule restriction ($restrictRuleSet()$).

5 EVALUATION

We evaluate our tag-based algorithm by comparing it to Motik et al.'s incremental reasoning (IR) algorithm based on DRed, described in the Delta-Reasoner [13]. We chose to compare those algorithms on the same implementation rather than comparing them on different applicative solutions: our goal is not to provide the fastest reasoner, but rather to offer an optimal solution for maintaining datasets incrementally in Web browsers using JavaScript. We evaluate these algorithms for ontology classification and initial insertion, insertion of new triples, deletion, insertion of known triples and selection. The three latter represent Web applications cycles as illustrated in Section 2. We run each algorithm in Google Chrome v.54.0.2840.99, on a Lenovo Ideapad 700-15ISK (Intel Core i5-6300HQ @2.3GHz - 4GB RAM).

[5] http://pellet.owldl.com/
[6] https://www.cs.ox.ac.uk/isg/tools/RDFox/
[7] http://chrjs.net/
[8] https://www.npmjs.com/package/hylar
[9] https://nodejs.org
[10] http://browserify.org/
[11] https://github.com/RubenVerborgh/SPARQL.js
[12] http://www.w3.org/TR/rdf-interfaces/#triples
[13] Rdfstore.js is a graph store implementation with support for SPARQL 1.0 and 1.1/Update. Available at https://github.com/antoniogarrote/rdfstore-js

5.1 Datasets and rules

We generated 3 datasets (O1, O2 and O3) using the Lehigh University Benchmark (LUBM) [5]. They are based on the Univ-Bench Ontology[14] schema, which has $\mathcal{ALEHI}+$ expressivity and contains 36 SubClassOf, 6 EquivalentClasses, 5 SubObjectPropertyOf, 1 TransitiveObjectProperty, 21 ObjectPropertyDomain, 18 ObjectPropertyRange and 4 DataPropertyDomain axioms, as well as 43 Class Assertions, 25 Object Property Assertions and 7 Data Property Assertions[15]. O1, O2 and O3 contain respectively 8824, 7394, and 5759 triples, and correspond to the initial insertion. The evaluation uses the same rule sets as in Section 4.

5.2 Practical evaluation

We ran 5 evaluation tasks: classification and initial dataset insertion (CLASSIF+INIT), insertion, deletion, re-insertion and selection for both IR and TB algorithms[16]. Inserted and deleted data have also been generated with LUBM and contain 500 triples. Each task applies the five rule sets described above. Our results are depicted in Figure 2. Processing times for each task are written in milliseconds. This table also shows the time difference betweeen IR and TB (Diff.), as well as the *performance* of TB (Perf.), i.e. the percentage of time gained if using our solution instead of IR, for a particular task. The "10 CYCLES" column sums the results for (i) classification and initial insertion, (ii) insertion, and ten cycles of (iii) deletion and (iv) re-insertion. Such cycles correspond to applicative scenarios such as the one described in Section 2.

Classification and initial insertion. As expected, TB maintenance does not outperform IR for these tasks, as it adds the cost of tagging facts. Although the number of rules and the size of the schema influence these results (e.g. O1 is more costly on TB as it is more expressive), RL profile reasoners do not target large classification tasks (an OWL-EL reasoner would probably be more suitable). Moreover, in Web applications, the classification and initial dataset insertion usually involve shared data and their results can therefore be computed on a server and cached for all clients. As tagging time is related to the number of triggered rules and their possible recursivity, its overhead is reduced for already closed datasets.

First insertion. Again, it takes longer for TB maintenance to perform a first insertion due to the additional tagging step. In this case, we can note that the instance number and the expressivity influences the results, as all facts - including instances - have to be tagged while firstly inserted in the ontology. Results show that this overhead varies according to the number of activated rules (*i.e.* the number and variety of OWL constructs).

Deletion. As expected, deletion is much (more than 50%) faster on TB maintenance, as IR over-deletion is replaced with a single iteration over the KB. Instance numbers significantly affect processing times in both algorithms.

Re-insertion. Re-inserting the same triples is also much faster on TB maintenance, as re-inserted triples do not have to be re-evaluated. TB performs particularly well with high expressivity (such as transitivity + inverse and equivalence rules). As in the deletion process, the number of instances is the most influential parameter.

Selection. Selections in IR are straightforward and give stable processing times. They are slower on TB maintenance as our algorithm checks the validity of each fact returned by the KB. With respect to IR, TB maintenance could then significantly impact SELECT queries with high numbers of triples or highly interrelated datasets. However, SELECT operations are much faster than the previous ones. Hence, despite its important value in percentage, this overhead sounds acceptable in terms of absolute times (about twenty milliseconds), as it only corresponds to a couple of frame rates of the most performant Web applications.

Multiple cycles. Here, we can see all the interest of our approach: the initial tagging cost at classification and first insertion is re-gained along deletions/re-insertion cycles. Due to space limitation, we only show figures for 10 cycles. However, for all situations in our evaluation, TB maintenance outperforms the given IR implementation from 4 cycles, and its gain in total computing time exceeds 50% for 100 cycles.

5.3 Evaluation synthesis

This evaluation shows that TB maintenance outperforms this implementation of the IR algorithm when data are being cyclicly deleted and reinserted into the reasoner, despite its cost on first insertions and selections. For applications going through hundreds of such cycles, TB maintenance can represent a massive performance improvement. This approach can particularly fit applications that rely on constantly changing data. For instance, context-aware applications that take adaptation decisions according to environmental (sensor) data can now integrate their own Web-based reasoner, process these data in Web clients and behave autonomously.

5.4 Discussion

Inserting a fact in a KB requires performing a transitive closure of the graph. The number of times a rule-based reasoner executes the rule evaluation loop depends on the data and on the expressivity of the used DL[17]. As tag-based is a maintenance approach, it does not aim at reducing the whole reasoning process complexity, but at performing it as rarely as possible. Hence, it can be considered as "storing" the reasoning complexity in causes to avoid recomputing it at deletions and rederivations. However, our evaluations show that in a common use case, it keeps affordable. We then propose a method to ensure its cost stay limited. In order to limit both the number of causes and the inflation of

[14] http://swat.cse.lehigh.edu/onto/univ-bench.owl

[15] Metrics provided by Protégé 5.0.0 - http://protege.stanford.edu/

[16] The TB fact forgetting algorithm that prevents KB inflation is not evaluated, as it is performed asynchronously during idle time.

[17] It is said to be EXPTIME-complete in $|KB|$ with $\mathcal{SHIQ}$ [9], and even untractable with other DLs [2] for tasks such as satisfiability or subsumption.

IR	CLASSIF+INIT			INSERT			DELETE			RE-INSERT			SELECT			10 CYCLES		
	O1	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*
Rsub	2 593	4 477	3 565	982	1 325	1 181	2 124	2 953	2 372	942	1 287	1 219	21	20	14	34 235	48 202	40 656
Rtrans-inv	2 224	4 129	3 204	1 139	969	767	1 561	2 155	1 592	1 110	924	743	21	15	15	30 073	35 888	27 321
Requiv	1 760	1 526	1 043	374	343	287	710	836	724	323	344	338	24	18	18	12 464	13 669	11 950
Requal	2 218	3 565	2 734	336	465	389	638	849	743	291	385	383	29	25	21	11 844	16 370	14 383
Rall	3 837	5 716	4 505	2 967	2 282	1 944	6 202	5 308	3 987	2 737	2 282	1 988	28	25	18	96 194	83 898	66 199
TB	CLASSIF+INIT			INSERT			DELETE			RE-INSERT			SELECT			10 CYCLES		
	O1	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*	*O1*	*O2*	*O3*
Rsub	5 121	4 650	3 617	1 313	1 333	1 179	827	738	665	798	656	675	39	30	25	22 684	19 923	18 196
Diff.	-2 528	-173	-52	-331	-8	2	1 297	2 215	1 707	144	631	544	-18	-10	-11	11 551	28 279	22 460
Perf. (%)	-97	-4	-1	-34	-1		**61**	**75**	**72**	**15**	**49**	**45**	-86	-50	-79	**34**	**59**	**55**
Rtrans-inv	4 302	4 222	3 305	1 156	1 028	802	674	589	533	676	649	558	41	28	27	18 958	17 630	15 017
Diff.	-2 078	-93	-101	-17	-59	-35	887	1 566	1 059	434	275	185	-20	-13	-12	11 115	18 258	12 304
Perf. (%)	-93	-2	-3	-1	-6	-5	**57**	**73**	**67**	**39**	**30**	**25**	-95	-87	-80	**37**	**51**	**45**
Requiv	1 782	1 597	1 048	375	344	360	154	187	222	180	215	247	41	32	30	5 497	5 961	6 098
Diff.	-22	-71	-5	-1	-1	-73	556	649	502	143	129	91	-17	-14	-12	6 967	7 708	5 852
Perf. (%)	-1	-5				-25	**78**	**78**	**69**	**44**	**38**	**27**	-71	-78	-67	**56**	**56**	**49**
Requal	4 050	3 812	2 877	408	480	454	160	164	241	175	168	267	46	29	33	7 808	7 612	8 411
Diff.	-1 832	-247	-143	-72	-15	-65	478	685	502	116	217	116	-17	-4	-12	4 036	8 758	5 972
Perf. (%)	-83	-7	-5	-21	-3	-17	**75**	**81**	**68**	**40**	**56**	**30**	-59	-16	-57	**34**	**54**	**42**
Rall	7 510	6 118	4 863	3 519	2 529	2 001	1 652	1 346	1 074	1 541	1 671	1 480	45	36	28	42 959	38 817	32 404
Diff.	-3 673	-402	-358	-552	-247	-57	4 550	3 962	2 913	1 196	611	508	-17	-11	-10	53 235	45 081	33 795
Perf. (%)	-96	-7	-8	-19	-11	-3	**73**	**75**	**73**	**44**	**27**	**26**	-61	-44	-56	**55**	**54**	**51**

Figure 2: Evaluation results

the knowledge base size (which is higher in TB maintenance as explicit facts are not removed), we suggest to limit the number of explicit facts. Our underlying hypothesis is that our reasoner targets Web applications that can run on small devices such as smartphones. It is not intended for storing application history but to receive facts that will trigger rules at the application level. In our scenario, the phone GPS coordinates are raw numeric values. They are not inserted "as is" in the reasoner but are transformed into facts that fit the application requirements (the phone is located in the house neighborhood). In these conditions, client-side reasoning can save both Web application developers' time while constructing their datasets (by using regular Semantic Web modeling tools), and bandwidth (by leaving saturation and decision processes up to the clients). Using a known set of explicit tags, TB reasoning allows application designers to first-insert and delete these facts at bootstrap or asynchronously, to pre-compute the tagging step and ensure performance at runtime.

6 RELATED WORK

6.1 OWL profiles and Web reasoning

OWL 2 profiles[18] help adjusting the trade-off between expressivity and efficiency. Each profile (EL, QL, RL) has its own specificities and targets different reasoning tasks. Reasoning tasks differ in terms of data, query and taxonomic complexity [12]. The choice of the appropriate OWL profile is crucial to reduce reasoning overheads, but not always sufficient as reasoners mostly rely on materialization (e.g. pre-compute and store inferences [14]) which is computationally intensive. EL is suitable for very large TBoxes and would not fit Web applications such as the one we describe in Section 2 as we expect their ABoxes to be heavier than their TBoxes. QL is appropriate for applications that manipulate high volumes of instances. It relies on query rewriting, which is not appropriate for Web applications that require fast query answering such as in our scenario. RL is more suitable, as it allows all axioms to be represented as logical implications and rules to be constructed as needed: to enable reasoning about OWL constructs, one can define both entailment rules corresponding to the expressive power expected for the application, and application-specific rules. RL reasoners can involve a large amount of explicit facts [11], and inferences are pre-computed and explicitly stored, so that queries can be answered simply by querying the store [8]. This makes this profile suitable Web applications that require flexibility and need fast query answering.

6.2 Incremental reasoning in RL

Web applications also need to handle numerous data updates. Reasoners embedded in those applications can then rely on IR [13] to avoid entire recomputations. Several improvements of IR currently exist. The *fact-dependency tracking* from [4] is similar to our *derivedFrom* tagging. However, unlike their solution, we track references of facts and provide improved query answering through validity checking in comparison to their query reformulation. The *counting* method [6] also tracks alternative derivations for each fact but does not support recursive rules. However, even *counting* algorithms that support recursion such as in [1] do not reduce the re-insertion cost as alternative derivations are not explicitly stated but rather counted. Nowadays, most incremental maintenance algorithms are based on Gupta et al.'s delete-rederive (DRed) [6]. DRed improves performance as it ensures

[18] http://www.w3.org/TR/owl2-profiles/

that the rules apply only to modified facts and thus prevents complete and successive recalculations of the KB on each update. The solution presented in [10] relies on DRed for the classification task, but it exclusively targets $\mathcal{EL}+$ ontologies with complex and changing TBoxes to tackle re-classification issues, which differs from our approach. In [15], Motik et al. tackle the derivation redundancy issue encountered in the overdeletion step using a semi-naive materialization approach that combines backward and forward (BF) chaining. This improvement however still relies on rule matching and evaluation at deletion, whereas our solution avoids overdeletion and re-derivation. They implemented this approach in the RDFox triplestore [16] that targets highly scalable applications. Yet, the HyLAR Reasoner is a JavaScript solution that targets Web browsers, which is currently not possible with RDFox. Hence, we did not compare those two reasoners. Nor did we compare BF with TB because the complex data structures used in BF (several rules bodies that would be duplicated into annotated queries) are not efficient in JavaScript (they actually perfom worse than regular IR). The Constraint Handling Rules [3] language also allows efficient rule-based reasoning and has been implemented in JavaScript. Unfortunately, it is not able to keep track of inferences, which prevents our fact-tagging approach to be used on this reasoner.

7 CONCLUSION

This paper addresses the issue of overdeleting and re-deriving facts in DRed Incremental Reasoning (IR). We propose a Tag-Based (TB) incremental maintenance approach that improves the DRed-based Incremental Reasoning (IR) overdeletion and re-derivation steps with three algorithms: implicit fact tagging, tag-based update and fact-filtering. The first two are executed to update the KB and the third to filter valid facts out of SELECT query results. Our approach targets Web applications that face multiple cycles of data deletions and reinsertions. Our complexity analysis shows that, to the initial cost of fact tagging at first insertion and validity assessment at selection, the complexity of re-insertion and deletion operations drops to linear, regardless of the reasoning conditions. Evaluation results show that the cost of TB maintenance is slightly higher for first insertions and for selections, but outperforms an implementation of IR at deletion and reinsertion. This cost is also re-gained within a few cycles.

In order to stimulate the adoption of semantic technologies in the Web community, it is implemented in the HyLAR reasoner, that proposes both IR and TB algorithms and is available as server-side (Node.js) or client-side (Bower) packages. HyLAR ships with a basic set of entailment rules that can be extended or reduced according to applications needs, and can be integrated with other tools, such as popular JavaScript frameworks, as well as our reasoning location adaptation framework [18].

As future work, we will extend the set of built-in OWL rules and stream reasoning capabilities to deal with larger datasets, and explore pattern mining techniques to discretize frequent sets of causes to reduce the KB size. We also plan to provide SWRL support in the reasoner to easily integrate it with authoring tools, such as Protégé[19], to foster Semantic Web application adoption.

ACKNOWLEDGEMENT

This work is supported by the French ANR (Agence Nationale de la Recherche) under the grant number <ANR-13-INFR-012>.

REFERENCES

[1] Hasanat M Dewan, David Ohsie, Salvatore J Stolfo, Ouri Wolfson, and Sushil Silva. 1992. Incremental database rule processing in PARADISER. *Journal of Intelligent Information Systems* 1, 2 (1992), 177–209.

[2] Francesco M Donini. 2003. Complexity of reasoning. In *The description logic handbook*. Cambridge University Press, 96–136.

[3] Thom Frühwirth. 2015. Constraint handling rules-what else?. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*. Springer, 13–34.

[4] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. 2013. Efficient query answering against dynamic RDF databases. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 299–310.

[5] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.

[6] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[7] Antonio Garrote Hernández. [n. d.]. A JavaScript RDF store and application library for linked data client applications. Citeseer.

[8] I Horrocks and PF Patel-Schneider. 2010. Knowledge Representation and Reasoning on the Semantic Web: OWL. (2010).

[9] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. 2005. Data complexity of reasoning in very expressive description logics. In *IJCAI*, Vol. 5. 466–471.

[10] Yevgeny Kazakov and Pavel Klinov. 2013. Incremental reasoning in OWL EL without bookkeeping. In *The Semantic Web–ISWC 2013*. Springer, 232–247.

[11] Markus Krötzsch. 2012. *OWL 2 Profiles: An introduction to lightweight ontology languages*. Springer.

[12] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, Carsten Lutz, et al. 2009. OWL 2 web ontology language: Profiles. *W3C recommendation* 27 (2009), 61. https://www.w3.org/TR/owl2-profiles/

[13] Boris Motik, Ian Horrocks, and Su Myeon Kim. 2012. Delta-reasoner: a semantic web reasoner for an intelligent mobile platform. In *Proceedings of the 21st international conference companion on World Wide Web*. ACM, 63–72.

[14] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Combining rewriting and incremental materialisation maintenance for datalog programs with equality. *arXiv preprint arXiv:1505.00212* (2015).

[15] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: the Backward/-Forward Algorithm. In *Proc. AAAI*.

[16] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. In *The Semantic Web - ISWC 2015*. Vol. 9367. Springer International Publishing, 3–20.

[17] Mehdi Terdjimi, Lionel Médini, and Michael Mrissa. 2015. HyLAR: Hybrid Location-Agnostic Reasoning. In *ESWC Developers Workshop 2015*. 1.

[18] Mehdi Terdjimi, Lionel Médini, and Michael Mrissa. 2016. HyLAR+: Improving Hybrid Location-Agnostic Reasoning with Incremental Rule-based Update. In *WWW'16: 25th International World Wide Web Conference Companion*.

[19] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. 2014. Web-Scale Querying through Linked Data Fragments. In *LDOW (CEUR Workshop Proceedings)*, Vol. 1184. CEUR-WS.org.

[19] http://protege.stanford.edu/