Chapter 3

Incremental Reasoning on RDF streams

Daniele Dell'Aglio

DEIB – Politecnico of Milano, P.za L. Da Vinci, 32. I-20133 Milano - Italy

Emanuele Della Valle

DEIB – Politecnico of Milano, P.za L. Da Vinci, 32. I-20133 Milano - Italy

3.1 Introduction

The introduction of stream processing methods in the Semantic Web enables the management of data streams on the Web. Chapter **??** introduced models for RDF stream and several extensions of SPARQL engines with windows for stream processing. The chapter assumes the absence of a TBox, so it is possible to compute the query answer without considering the ontology entailment defined through a TBox described in an ontological language. In this chapter, we relax this constraint and we consider the case of query answering over RDF streams when the TBox is not empty. In particular, we focus on Stream Reasoning [9], the topic that studies how to compute and incrementally maintain the ontological entailments in RDF streams.

In traditional Semantic Web reasoning data are usually static or quasi-static[1], so the whole computation of the ontological entailment can be executed every time the data change. When we consider RDF streams the static

[1]The data change so slowly to be treated as static data

hypothesis is not valid anymore: RDF stream engines work with highly dynamic data and they need to process them faster than new data arrives to avoid congestion states. In this scenario, traditional materialization techniques could fail; a possible solution is the incremental maintenance of the materialized entailment using adaptations of the classical DRed algorithm [7, 14]: when new triples are added, the deducible data is added to the materialization; similarly, when triples are deleted the triples that cannot be deducted anymore are removed from the entailment. The idea of incremental maintenance was previously delivered in the context of deductive databases, where logic programming were used for the incremental maintenance of such entailments. The idea of incrementally maintain an ontological entailment was proposed first by [15]: in this work the authors propose a version of DRed based on logic program that computes the changes in the ontological entailment, and consequently computes the new materialization adding and removing the two delta sets.

In this chapter, we present IMaRS (Incremental Materialization for RDF Streams) [5], a variation of DRed for the incremental maintenance of the window materializations. In general, the main problems of the incremental maintenance are the deletions: it is a complex task to determine which consequences are not valid anymore when statements are removed by the knowledge base. IMaRS exploits the nature of RDF streams in order to cope with this problem. As we show, the window operators allow to determine when a statement will be deleted from the materialization. IMaRS, when triples are inserted in the window, computes when they will be deleted and annotates them with an expiration time stamp. This allows IMaRS to work out a new complete and correct materialization whenever a new window of RDF streams arrives by dropping explicit statements and entailments that are no longer valid. We provide experimental evidence that our approach significantly reduces the time required to maintain a materialization at each window change, and opens up for several further optimizations.

In the rest of the chapter, we first introduce in Section 3.2 some basic concepts. Then, we present a running example based on a social network stream in Section 3.3. Section 3.4 presents IMaRS, with the description of the algorithm. A list of related works is presented in Section 3.5; finally, we evaluate IMaRS in Section 3.6 and close with some future direction in Section 3.7.

3.2 Basic concepts

In this section we introduce the idea behind the RDF streams and RDFS+, the ontological language we consider in our work.

3.2.1 RDF stream

Even if the notion of RDF streams and RDF stream engines are widely covered in Chapter **??**, we briefly summarize the main concepts useful to understand this chapter. A **RDF stream** [6] is an infinite sequence of timestamped RDF triples ordered by their timestamps. Each **timestamped triple** is a pair constituted by an RDF triple and its timestamp τ.

$$\begin{aligned} &\dots \\ &< subj_{i+1}, pred_{i+1}, obj_{i+1} >: [\tau_{i+1}] \\ &< subj_i, pred_i, obj_i >: [\tau_i] \\ &\dots \end{aligned}$$

A **timestamp** (or **application time**) τ is a natural number and it represents the time associated to the RDF triple. They are monotonically non-decreasing in the stream ($\tau_i \leq \tau_{i+1}$); they are not strictly increasing to allow for expressing contemporaneity, i.e., a stream can contain two or more RDF triples with the same application time. Thus, timestamps are not required to be unique.

The systems able to process and query RDF stream are called **RDF stream engines**. The general idea to process an infinite sequence of elements is to use several operators to extract portions of the stream and to work on them. One of the most famous operator is the **window** [4]. A window contains the most recent elements of the stream and its contents are updated over time (the window *slides* over the stream).

3.2.2 RDFS+

RDFS+ [1] is an extension of RDFS (Section **??**) with additional elements of OWL, such as transitive properties and inverse properties. Table 3.1 summarizes the elements of RDFS+. This language was defined before OWL2 (see **??**) and it aimed to become a good trade-off between RDFS and OWL-DL (see **??**): on the one hand it overcomes the limited expressiveness of RDFS and on the other one it performs faster than OWL-DL. Nowadays RDFS+ language expressiveness can be considered as a subset of OWL2 RL (Section **??**). The language is supported by several systems, such as AllegroGraph[2] and SPIN[3].

[2] Cf. `http://www.franz.com/agraph/support/learning/Overview-of-RDFS++.lhtml`
[3] Cf. `http://topbraid.org/spin/rdfsplus.html`

TABLE 3.1: RDFS+ elements

Rule ID	Body	Head
rdf1	?s ?p ?o	⇒ ?p a rdf:Property
rdfs2	?p dom ?c . ?x ?p ?y	⇒ ?x a ?c
rdfs3	?p rng ?c . ?x ?p ?y	⇒ ?y a ?c
rdfs5	?p1 sP ?p2 . ?p2 sP ?p3	⇒ ?p1 sP ?p3
rdfs7	?p1 sP ?p2 . ?x ?p1 ?y	⇒ ?x ?p2 ?y
rdfs9	?c1 sC ?c2 . ?c2 sC ?c3	⇒ ?c1 sC ?c3
rdfs11	?c1 sC ?c2 . ?x a ?c1	⇒ ?x a ?c2
prp-trp	?p a TP . ?x ?p ?y . ?y ?p ?z	⇒ ?x ?p ?z
prp-inv1	?p1 inv ?p2 . ?x ?p1 ?y	⇒ ?y ?p2 ?x
prp-inv2	?p1 inv ?p2 . ?x ?p2 ?y	⇒ ?y ?p1 ?x
eq-sym	?x sA ?y	⇒ ?y sA ?x

3.3 Running Example

The goal of IMaRS is the processing of the materialization of an ontological entailment of the actual content of the active window and its incremental maintenance across time. Usually, when the content of the window changes, a set of triples is removed and another set of triples is added. The consequences on the ontological entailments are therefore twofold: on the one hand there can be inferred triples that are not valid anymore – triples derived by deleted triples, on the other hand there are new inferable triples – triples that can be derived by the new added data.

As running example, we refer to a scenario of stream processing over a stream of posts of a social network, e.g. Twitter[4]. We consider a simple scenario of a stream of published posts and each post is created by an author. The TBox $\mathcal{T}$ is defined through SIOC; there two classes, `sioc:UserAccount` representing the authors and `sioc:Post` representing the messages. Authors and posts are related through a relation `sioc:creator_of` and its inverse property `sioc:has_creator`. The formalization of $\mathcal{T}$ is:

$$sioc{:}UserAccount \sqsubseteq \top$$
$$sioc{:}Post \sqsubseteq \top$$
$$sioc{:}creator\_of \equiv sioc{:}has\_creator^{-}$$
$$ran(sioc{:}has\_creator) \sqsubseteq sioc{:}UserAccount$$

Consequently, the serialization of $\mathcal{T}$ in N3 RDF is:

[4]Cf. `http://www.twitter.com/`

```
@prefix sioc: <http://rdfs.org/sioc/ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
sioc:UserAccount a owl:Class .
sioc:Post a owl:Class .
sioc:creator_of a owl:ObjectProperty .
sioc:creator_of owl:inverseOf sioc:has_creator .
sioc:has_creator rdfs:range sioc:UserAccount .
```

Let's suppose we are interested in retrieving the list of the active users in the last 5 minutes. An active user is a user that created at least one post in the previous minutes. We can represent the query through C-SPARQL in the following way:

```
PREFIX sioc: <http://rdfs.org/sioc/ns#>
REGISTER QUERY active_users AS
SELECT DISTINCT ?author
FROM STREAM <http://example.org/stream> [RANGE 5m STEP 1m]
WHERE{
     ?author a sioc:UserAccount;
             sioc:creator_of ?post
} GROUP BY ?author
```

Let's consider now the scenario represented in Figure 3.1. At time $\tau_1 = 10$ (Figure 3.1(a)) two RDF timestamped triples of the stream $\mathbb{S}$ are in the active window W_1(whose scope is $[5, 10)$):

```
1 <:Adam sioc:creator_of :tweet1>:[5]
2 <:Bob sioc:creator_of :tweet2>:[7]
```

In the following, we indicate with t_i the triple at line i. Triples in the window are not enough to compute an non-empty answer of the query. Anyway, the query can be answered if we consider the materialization obtained exploiting the axioms in $\mathcal{T}$ from which several triples can be derived:

```
3 <:tweet1 sioc:has_creator :Adam>
4 <:tweet2 sioc:has_creator :Bob>
5 <:Adam  a sioc:UserAccount>
6 <:Bob  a sioc:UserAccount>
```

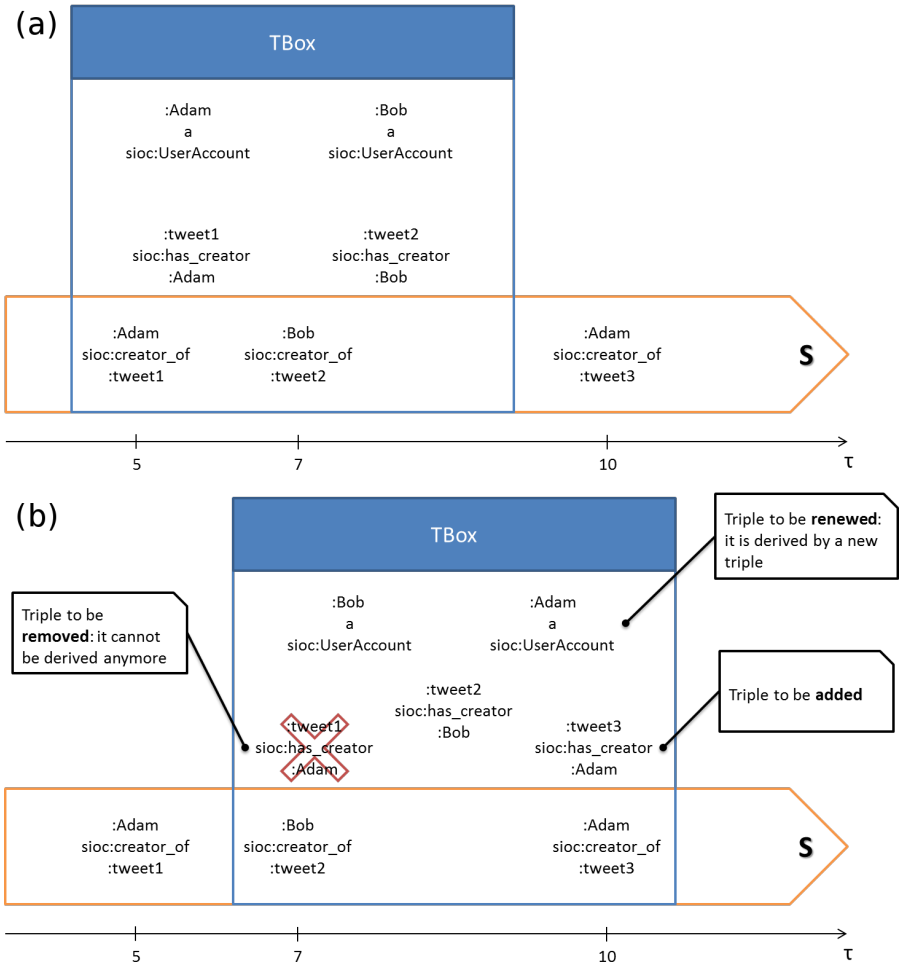

FIGURE 3.1: Example of incremental derivation. (a) shows the IMaRS window W_1 at τ_1; (b) shows the IMaRS window W_2 at τ_2. In (b) the differences in the materialization are highlighted: they are the changes that the incremental maintenance algorithm should apply to compute the correct materialization.

Taking into account the materialization (triples $t_1 \dots t_6$) the answer is the following:

- `:Adam`
- `:Bob`

At time $\tau_2 = 11$ the content of the window W_2 (whose scope is $[6, 11)$) changes (Figure 3.1(b)): t_1 expires, so it is deleted from the window, while the triple t_7 is added to the window:

```
7 <:Adam sioc:creator_of :tweet3>:[10]
```

The triples of the stream $\mathbb{S}$ in the active windows are now t_2 and t_7. The ontology entailment is not valid anymore and has to be updated in the following way:

- triple t_3 has to be *removed*: it was derived from t_1;
- triples t_4 and t_6 are *maintained*: they are derived from t_7 and it is again in the window;
- a new triple is *added* to the materialization:

```
8 <:tweet3 :has_creator :Adam>
```

  because it is derived from t_7 and $\mathcal{T}$;
- triple t_5 is *renewed*: it is not derivable anymore from t_1 but it can be inferred from t_7 and $\mathcal{T}$.

3.4 Incremental Materialization algorithm for RDF Streams

In this section, we present IMaRS and we explain how it helps in achieve the behavior described above.

3.4.1 Assumptions and preliminary definitions

In the following, we introduce the definitions at the basis of the algorithm presented above, and the assumptions until which the IMaRS algorithm works:

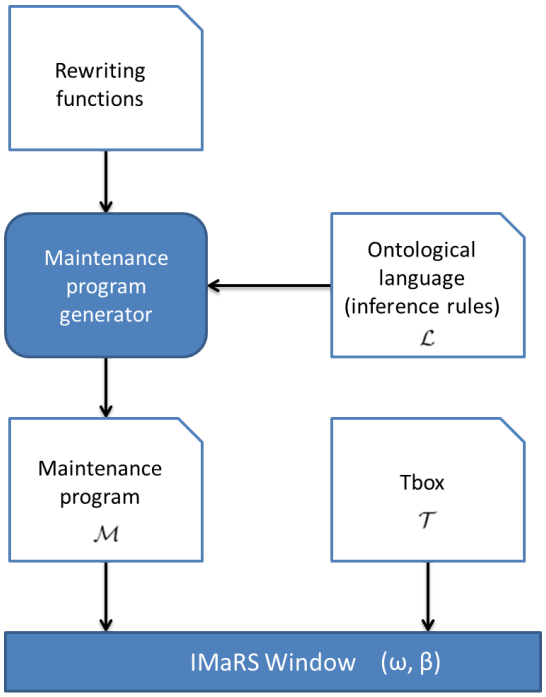

FIGURE 3.2: Creation of the maintenance program

the use of windows to maintain the materialization, RDFS+ as ontological language, and the absence of TBox assertion in the streams.

The computation of the materialization in an RDF stream engine is strictly related to the window operator. The triples of the stream that have to be considered for the materialization at a given time τ are those contained in the scope of the active window at that time.

We define **IMaRS window** a time-based sliding window that can maintain the materialization of the ontological entailment through IMaRS. A IMaRS window has four parameters:

$$W_{IMaRS}(\omega, \beta, \mathcal{T}, \mathcal{M}) \tag{3.1}$$

The two parameters ω and β are the same used in time-based sliding windows [4]: ω is the size of the window and β is the slide parameter. $\mathcal{T}$ is the TBox describing the model of the stream and $\mathcal{M}$ is the maintenance program.

A **maintenance program** $\mathcal{M}$ is the logic program that is executed over the window content. It is composed by a set of rules (**maintenance rules**) required to compute the two sets of triples Δ^+ and Δ^- that respectively have to be added and removed to maintain the materialization.

The process of creation of the maintenance program $\mathcal{M}$ and the IMaRS

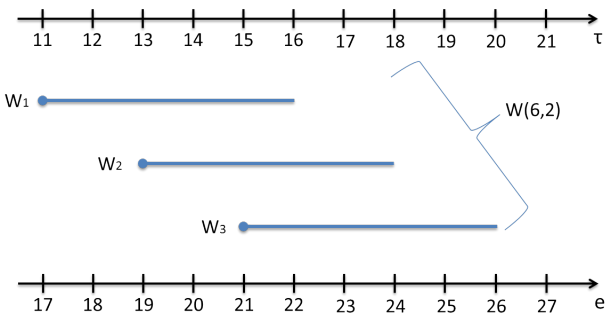

FIGURE 3.3: Assignment of the expiration time to triples in $\mathbb{S}$

window W_{IMaRS} is depicted in Figure 3.2. The maintenance program $\mathcal{M}$ is derived by the maintenance program generator: is takes as input a set of rewriting rules and an ontological language $\mathcal{L}$ (e.g., RDFS+) expressed as a set of inference rules. As explained in Section 3.4.2, for each inference rule, the generator produces one or more maintenance rules and prunes part of them exploiting the assumptions presented in this section.

The maintenance program $\mathcal{M}$, and the TBox $\mathcal{T}$ (expressed through $\mathcal{L}$) are then used as one of input to create a IMaRS window.

An important notion at the core of our approach is the **expiration time** e. It indicates the time at which a triple will not be valid anymore and it can be consequently be removed from the window. Each triple in the IMaRS window has an expiration time, both triples in the stream $\mathbb{S}$ and inferred ones. Recalling the fact that a triple of the stream $\mathbb{S}$ is a timestamped triple $<s, p, o>:[\tau]$, it is worth noting that the expiration time e is an additional timestamp and it is significantly related to τ. We indicate a triple $t = <s, p, o>$ with application time τ and expiration time e in the following way: $<s, p, o, e>:[\tau]$ (or shortly $t^e:[\tau]$). The expiration time e for a triple t is set in the following way:

1. if the triple $t:[\tau]$ is in the stream $\mathbb{S}$, then its expiration time is set to $e = \tau + \omega$ (where ω is the window width), because by that time the triple will no longer be in the scope of the window;

2. if the triple t is derived by a set of triples $<s_1, p_1, o_1, e_1>, \ldots, <s_n, p_n, o_n, e_n>$ in the actual materialization of the window W and by a set of triples $d_1, \ldots, d_m$ of the TBox $\mathcal{T}$, then its expiration time is set to $e = \min\{e_1, \ldots, e_n\}$, because the deduction will no longer hold as soon as one of the triples, which it is derived from, expires.

For the sake of comprehension, in the following, we omit the application time τ and we only use $<s, p, o, e>$ to indicate that the triple $<s, p, o>$ has an expiration time e.

The introduction of the expiration time introduces the possibility that the

IMaRS window will contain the same triple several times. We define a triple $t_1^{e_1}$ **duplicate** of $t_2^{e_2}$ if $t_1 = t_2$ (the two triples have the same subject, predicate and object).

Exploiting this definition, we can now define the notions of **renewal** and **most recentness**. A triple $t_1^{e_1}$ **is renewed by** a triple $t_2^{e_2}$ if they are duplicates and $e_2 > e_1$. Given a set of duplicates $D = \{t^{e_1}, \ldots, t^{e_n}\}$, $t^e \in D$ is the **most recent** triple if it has the latest expiration time, i.e. it does not exist a triple $t^{e'} \in D$ such that $e' > e$.

Duplicate triples in the window usually do not affect the query answering process; they could at most influence the number of returned results: in a case of a SELECT query the answer could contain several duplicate tuples. On the other hand it is mandatory to consider the most recent triple of a duplicate set when maintaining the materialization: in fact the inference process should take into account the most recent triples to assign the correct expiration time to new triples and consequently to preserve the completeness of the ontological entailment.

An important assumption of IMaRS is about the ontological language, the algorithm can be proved to correctly maintain the ontological entailments of a knowledge base expressed in RDFS+ (and consequently its subsets e.g., RDFS). At the moment IMaRS does not work with the OWL-RL language; we plan to extend the algorithm to support OWL-RL in our future works, as we discuss in Section 3.7.

The TBox $\mathcal{T}$ is considered to be static knowledge and we assume it is always valid. It means that $\mathcal{T}$ is part part of the materialization (IMaRS takes $\mathcal{T}$ into account in the derivation process) and its axioms do not expire. We denote this fact associating expiration time $e = \infty$ to TBox statements. We additionally assume that the input stream $\mathbb{S}$ cannot contain triples that extend or alter $\mathcal{T}$. We discuss in Section 3.7 the problem of relaxing those constraints.

3.4.2 Maintenance program generation

In this section, we explain the process that generates the maintenance program $\mathcal{M}$ given an ontological language $\mathcal{L}$. The ontological language $\mathcal{L}$ is defined by a set of inference rules.

As we explained above in the previous section, the task of $\mathcal{M}$ is the computation of two sets of triples Δ^+ and Δ^- that contain respectively the triples that should be added to the materialization and those that should be removed in order to maintain the ontological entailment correct and complete. $\mathcal{M}$ uses several graphs (namely, **contexts**) in order to compute Δ^+ and Δ^-. Table 3.2 introduces the contexts used by the IMaRS maintenance program. In addition to Δ^+ and Δ^-, IMaRS uses:

- the Mat context to store the actual materialization;

- the Ins context to store the triples of $\mathbb{S}$ that enter the window and triples derived by them;
- the New context to store the candidate triples to be added to the materialization;
- the Ren context to store duplicate triples that are renewed.

We denote the fact that an RDF triple $t = <s, p, o>$ with expiration time e is contained in a context $\mathcal{C}$ with a quintuple using the two following notations:

- a long version – $<s, p, o, e, \mathcal{C}>$;
- a short version – $t^e{:}\mathcal{C}$.

Given the sets I (IRIs), B (blank nodes), L (literals), C (contexts) and $N \subset L$ (natural numbers) we define $<s, p, o, e, \mathcal{C}>$ as a member of $(I \cup B) \times I \times (I \cup B \cup L) \times N \times C$. In a similar way, we extend the concept of triple pattern tp [12] to take into account the expiration time and the context. An extended triple pattern is a member of the set: $(I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V) \times (N \cup V) \times C$ (where V is the set of variables). We indicate with the notation $tp^e : \mathcal{C}$ the set of triples in $\mathcal{C}$ that satisfies tp.

We use these two notations to define the maintenance rules. A maintenance rule δ is a rule[5] in the form:

$$tp_1^{e_1}{:}\mathcal{C}_1 \,.\, tp_2^{e_2}{:}\mathcal{C}_2 \,.\, \ldots \,.\, tp_n^{e_n}{:}\mathcal{C}_n \,.\, f_b(e_1, \ldots, e_n, now) \Rightarrow tp^e{:}\mathcal{C} \tag{3.2}$$

The body of the rule is a conjunction of **conditions** $tp_x^{e_x}{:}\mathcal{C}_x$ and a boolean function f_b. A condition is satisfied if there is a mapping μ_x that, when applied to $tp_x^{e_x}$, allows to obtain a triple $t_x^{e_x}$ in $\mathcal{C}_x$. Consequently the body is satisfied if there exists at least a mapping μ that allows to satisfy each condition. f_b is a boolean function used to specify constraints on the expiration times of the involved triples; the function uses one or more expiration time stamps in $e_1, \ldots, e_n$ and a built-in now. now returns the time stamp of the update of the window contents (when the last window closes). It is worth to note that now could not change during the execution of $\mathcal{M}$; if it happens it means that the system is overloaded.

The head $tp^e{:}\mathcal{C}$ indicates that if the body is satisfied by mappings $Mp = \mu_1, \ldots, \mu_m$, then the triples $t_1^e, \ldots, t_m^e$ obtained applying each mapping in Mp to tp^e are added to $\mathcal{C}$. Expiration time e is computed through a function that takes as parameters the expiration time stamps of the conditions (e.g., the min function).

The maintenance rules can be easily rewritten in different languages; for

[5]The rule format is similar to the inference rules defined in **??**, extended with the expiration time stamps, the context notions and a boolean function f_b expressed on expiration time stamps.

TABLE 3.2: Contexts used by the maintenance program

Name	Content
Mat	Current materialization
Δ^+	Net additions required to maitain the materialization
Δ^-	Net deletions required to maitain the materialization
Ins	Set of timestamped triples in $\mathbb{S}$ inserted in the window in the last update and their derivations
New	Support context to compute the triples that should be added to the materialization
Ren	Set of triples that are renewed

example δ can be rewritten in Datalog using the contexts $\mathcal{C}_x$ as Datalog predicates and the other four elements (s_x, p_x, o_x and e_x)[6] as their arguments:

$$\mathcal{C}(s,p,o,e) : -\mathcal{C}_1(s_1,p_1,o_1,e_1),\ldots,\mathcal{C}_n(s_n,p_n,o_n,e_n), f_b(e_1,\ldots,e_n,now). \tag{3.3}$$

The maintenance program $\mathcal{M}$ is derived by the maintenance program generator; $\mathcal{M}$ is composed by two sets of rules, M_1 and M_2. The former is constant while the latter depends on the input ontological language $\mathcal{L}$. M_1 defines the relations among the contexts of $\mathcal{M}$ and the conditions under which it is possible to add a triple t^e in a context C; this set of rules is proposed in Table 3.3.

As we explain below, triples in the New and Ins contexts are used to compute new triples that can be added to the materialization, adding them to the Ins context. Rules δ_1^{Old} and δ_2^{Old} determine which are the renewed triples (triples for which there is a most-recent triple in Ins); those triples will be put in the Ren context. δ_1^{Old} looks for renewed triples in Mat, while δ_2^{Old} works on Ins. Rule δ_1^{New} puts in New all the triples of the materialization (Mat) with valid expiration time (expiration time greater than the actual one, i.e., now). RDF triples of the TBox are always moved in New, due to the fact they have expiration time ∞. δ_2^{New} puts in the New context the triples of Ins that are not renewed (i.e., are not in the Ren context). δ_1^- and δ_2^- contain the conditions until which triples can be added to Δ^-: the former selects the RDF triple expired, while the latter selects the renewed triples (i.e., i collects all the duplicates generated in the maintenance program). Finally the delta context Δ^+ is computed through the two rules δ^{++} and δ^+; a support context, Δ^{++} is used in the computation. It is a temporary context that contains triples in the New context but not in Mat (with different expiration time). Then, the difference between Δ^{++} and Ren defines Δ^+.

The set of maintenance rules in M_2 is derived by the ontological language

[6]Symbols s_x, p_x, o_x and e_x could be either variables or values

TABLE 3.3: Maintenance rules in M_1

Name	Maintenance Rule
δ_1^{New}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Mat> \ . (\mathtt{e} \geq \mathtt{now})$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, New>$
δ_2^{New}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ins> \ . \neg <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ren>$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, New>$
δ_1^{Old}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}_1, Ins> \ . <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Mat> \ . (\mathtt{e}_1 > \mathtt{e})$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ren>$
δ_2^{Old}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}_1, Ins> \ . <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ins> \ . (\mathtt{e}_1 > \mathtt{e})$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ren>$
δ_1^{-}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Mat> \ . (\mathtt{e} < \mathtt{now})$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, \Delta^{-}>$
δ_2^{-}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ren>$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, \Delta^{-}>$
δ^{++}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, New> \ . \neg <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}_1, Mat>$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, \Delta^{++}>$
δ^{+}	$<?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, \Delta^{++}> \ . \neg <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}_1, Ren>$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, \Delta^{+}>$

$\mathcal{L}$ through a **rewriting function** $\delta^{Ins}(ir)$:

Name	Rewriting function
$\delta^{Ins}(ir)$	$\{<?\mathtt{s}_1, ?\mathtt{p}_1, ?\mathtt{o}_1, ?\mathtt{e}_1, New> \ . <?\mathtt{s}_{i-1}, ?\mathtt{p}_{i-1}, ?\mathtt{o}_{i-1}, ?\mathtt{e}_{i-1}, New> \ .$ $<?\mathtt{s}_i, ?\mathtt{p}_i, ?\mathtt{o}_i, ?\mathtt{e}_i, Ins> \ . <?\mathtt{s}_{i+1}, ?\mathtt{p}_{i+1}, ?\mathtt{o}_{i+1}, ?\mathtt{e}_{i+1}, New> \ .$ $\ldots . <\mathtt{s}_n, \mathtt{p}_n, \mathtt{o}_n, ?\mathtt{e}_n, New> \ . ?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}, ?\mathtt{e}_n\}$ $\Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}, ?\mathtt{e}, Ins>\}$

This function is applied to each inference rule of $\mathcal{L}$ to generate the maintenance rules composing M_2. Given an inference rule $ir \in \mathcal{L}$:

$$ir : ?\mathtt{s}_1 \ ?\mathtt{p}_1 \ ?\mathtt{o}_1 \ . \ \ldots \ . \ ?\mathtt{s}_n \ ?\mathtt{p}_n \ ?\mathtt{o}_n \ \Rightarrow ?\mathtt{s} \ ?\mathtt{p} \ ?\mathtt{o}$$

the rewriting function generates n maintenance rules, where n is the number of conditions in the body of ir.

Considering the running example in Section 3.3, let's now derive its maintenance program $\mathcal{M}_{ex}$. The TBox $\mathcal{T}$ of the example is written in RDFS+, so this ontological language is the one that the maintenance program generator should consider to produce $\mathcal{M}_{ex}$. For the sake of brevity, we take into account the three RDFS+ rules `rdfs2`, `prp-inv1` and `prp-trp`[7] (see Table 3.1). As explained above, $\mathcal{M}_{ex}$ is the union of M_1 and M_2; M_2 is derived applying the rewriting function δ^{Ins} to every inference rule of RDFS+ and the result is reported in Table 3.4. Let's consider the inference rule `prp-trp`:

$$prp-trp : ?\mathtt{p} \ \mathtt{a} \ \mathtt{owl{:}TransitiveProperty} \ . \ ?\mathtt{x} \ ?\mathtt{p} \ ?\mathtt{y} \ . \ ?\mathtt{y} \ ?\mathtt{p} \ ?\mathtt{z} \ \Rightarrow ?\mathtt{s} \ ?\mathtt{p} \ ?\mathtt{o}$$

[7]Even if the transitive rule is not useful for the running example, we consider it as an example of rule with a body composed by more than two conditions.

TABLE 3.4: M_2 for a portion of RDFS+

$<?\mathtt{p}, \mathtt{rdfs{:}domain}, ?\mathtt{c}, ?\mathtt{e}_1, New> . <?\mathtt{x}, ?\mathtt{p}, ?\mathtt{y}, ?\mathtt{e}_2, Ins> .$
$?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2\}$
$\Rightarrow <?\mathtt{x}, \mathtt{rdf:type}, ?\mathtt{c}, ?\mathtt{e}, Ins>$
$<?\mathtt{p}, \mathtt{rdfs{:}domain}, ?\mathtt{c}, ?\mathtt{e}_1, Ins> . <?\mathtt{x}, ?\mathtt{p}, ?\mathtt{y}, ?\mathtt{e}_2, New> .$
$?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2\}$
$\Rightarrow <?\mathtt{x}, \mathtt{rdf:type}, ?\mathtt{c}, ?\mathtt{e}, Ins>$
$<?\mathtt{p1}, \mathtt{owl{:}inverseOf}, ?\mathtt{p2}, ?\mathtt{e}_1, New> . <?\mathtt{x}, ?\mathtt{p1}, ?\mathtt{y}, ?\mathtt{e}_2, Ins> .$
$?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2\}$
$\Rightarrow <?\mathtt{y}, ?\mathtt{p2}, ?\mathtt{x}, ?\mathtt{e}, Ins>$
$<?\mathtt{p1}, \mathtt{owl{:}inverseOf}, ?\mathtt{p2}, ?\mathtt{e}_1, Ins> . <?\mathtt{x}, ?\mathtt{p1}, ?\mathtt{y}, ?\mathtt{e}_2, New> .$
$?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2\}$
$\Rightarrow <?\mathtt{y}, ?\mathtt{p2}, ?\mathtt{x}, ?\mathtt{e}, Ins>$
$<?\mathtt{p}, \mathtt{rdf{:}type}, \mathtt{owl{:}TransitiveProperty}, ?\mathtt{e}_1, New> .$
$<?\mathtt{x}, ?\mathtt{p}, ?\mathtt{y}, ?\mathtt{e}_2, Ins> . <?\mathtt{y}, ?\mathtt{p}, ?\mathtt{z}, ?\mathtt{e}_3, Ins> .?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2, ?\mathtt{e}_3\}$
$\Rightarrow <?\mathtt{y}, ?\mathtt{p2}, ?\mathtt{x}, ?\mathtt{e}, Ins>$
$<?\mathtt{p}, \mathtt{rdf{:}type}, \mathtt{owl{:}TransitiveProperty}, ?\mathtt{e}_1, Ins> .$
$<?\mathtt{x}, ?\mathtt{p}, ?\mathtt{y}, ?\mathtt{e}_2, New> . <?\mathtt{y}, ?\mathtt{p}, ?\mathtt{z}, ?\mathtt{e}_3, Ins> .$
$?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2, ?\mathtt{e}_3\}$
$\Rightarrow <?\mathtt{y}, ?\mathtt{p2}, ?\mathtt{x}, ?\mathtt{e}, Ins>$
$<?\mathtt{p}, \mathtt{rdf{:}type}, \mathtt{owl{:}TransitiveProperty}, ?\mathtt{e}_1, Ins> .$
$<?\mathtt{x}, ?\mathtt{p}, ?\mathtt{y}, ?\mathtt{e}_2, Ins> . <?\mathtt{y}, ?\mathtt{p}, ?\mathtt{z}, ?\mathtt{e}_3, New> .$
$?\mathtt{e} = \mathtt{min}\{?\mathtt{e}_1, ?\mathtt{e}_2, ?\mathtt{e}_3\}$
$\Rightarrow <?\mathtt{y}, ?\mathtt{p2}, ?\mathtt{x}, ?\mathtt{e}, Ins>$

The body of `prp-trp` is composed by three conditions, so δ^{Ins} will generate three maintenance rules. These rules cover all the possible cases on which the derivation could be executed: one of the three triples in the body is added in the materialization (in the Ins context) and the other two ones are in the New context (candidate triples to be added to the new materialization). It is worth to note that the maintenance rule δ_2^{New} of M_1 moves triples from Ins to New: this allows to perform new derivations from derived triples (and not only by triples of $\mathbb{S}$). The first rules of $\delta^{Ins}(\mathtt{prp-trp})$ states that if the Ins context contains a triple that satisfies the triple pattern `<?p,a,owl:TransitiveProperty,`e_1`>` and the New context contains triples that satisfies `<?x,?p,?y,`e_2`>` and `<?y,?p,?z,`e_3`>`, then a new triple `<?x,?p,?z>` with expiration time $e = \mathtt{min}\{e_1, e_2, e_3\}$ will be added in the Ins context. The other two rules are similar.

One of the assumptions done in Section 3.4.1 is that the stream $\mathbb{S}$ does not modify the TBox $\mathcal{T}$. It means that $\mathbb{S}$ can not contains TBox triples such as `<?p1,owl:inverseOf,?p2>`; thus it is possible to optimize M_2 in the following way:

TABLE 3.5: Optimized M_2 (M_2^+) for RDFS+

Name	Maintenance Rule
δ_1^{Ins}	<?p, rdfs:domain, ?c, ∞,New> . <?x, ?p, ?y, ?e,Ins> $\Rightarrow$<?x, rdf:type, ?c, ?e,Ins>
δ_2^{Ins}	<?p1, owl:inverseOf, ?p2, ∞,New> . <?x, ?p1, ?y, ?e,Ins> $\Rightarrow$<?y, ?p2, ?x, ?e,Ins>
δ_3^{Ins}	<?p, rdf:type, owl:TransitiveProperty, ∞,New> . <?x, ?p, ?y, ?e_1,New> . <?y, ?p, ?z, ?e_2,Ins> .?e = min{?e_1, ?e_2} $\Rightarrow$<?y, ?p2, ?x, ?e,Ins>
δ_4^{Ins}	<?p, rdf:type, owl : TransitiveProperty, ∞,New> . <?x, ?p, ?y, ?e_1,Ins> . <?y, ?p, ?z, ?e_2,New> .?e = min{?e_1, ?e_2} $\Rightarrow$<?y, ?p2, ?x, ?e,Ins>

- delete the rules where a TBox triple is in the Ins context – it is not possible to have that case;
- rewrite the expiration time assignment ?e = min{?e_1, ..., ?e_n} replacing ∞ with ?e_i associated to TBox triples.

Table 3.5 shows the optimized maintenance program M_2^+, derived by M_2 through the application of the optimizations described above. It is possible to observe that the new program is composed by a lower number of maintenance rules (4 instead of 7). The maintenance rules of M_2^+ are simpler than the ones of M_2: when the expiration time ?e_t associated to a TBox triple t is assigned to ∞ it is easy to observe that: $e = \min\{e_t, e_1, \ldots, e_n\} = e = \min\{e_1, \ldots, e_n\}$. Additionally, if there are only two expiration timestamps, the assignment $e_t = \infty$ implies $e = \min\{e_t, e_1\} = e_1$.

It is now possible to replace M_2 with M_2^+ and the maintenance program for the running example is $\mathcal{M}_{ex}^+ = M_1 \cup M_2^+$.

3.4.3 Execution of the maintenance program in the IMaRS window

As explained above, the maintenance program $\mathcal{M}$ is used by the IMaRS window to compute the ontological entailment and to maintain it correct and complete. Every time the content of the window changes, the maintenance rules run over the actual materialization to compute the new one. The main steps of the materialization maintenance are:

- the actual materialization is in the Mat context;
- triples of $\mathbb{S}$ entering the windows are annotated with their expiration time and are put in the Ins context;

- the maintenance program is executed: rules are executed in a order determined by the dependencies among them, as happens in $Datalog^{\neg}$ (Datalog with stratified negation);

- the materialization is updated adding the content of Δ^+ and removing the content of Δ^- (i.e. $Mat \cup \Delta^+ \setminus \Delta^-$).

Let's consider the running example to show how the IMaRS window works to obtain the behaviour described in Section 3.3. Just to summarize, at $\tau_1 = 10s$ the materialization Mat of the IMaRS window W_{IMaRS} is:

```
1  <sioc:UserAccount a owl:Class,∞>
2  <sioc:Post a owl:Class,∞>
3  <sioc:creator_of a owl:ObjectProperty,∞>
4  <sioc:creator_of owl:inverseOf sioc:has_creator,∞>
5  <sioc:has_creator rdfs:range sioc:UserAccount,∞>
6  <:Adam sioc:creator_of :tweet1,10>:[5]
7  <:Bob sioc:creator_of :tweet2,12>:[7]
8  <:tweet1 sioc:has_creator :Adam,10>
9  <:tweet2 sioc:has_creator :Bob,12>
10 <:Adam  a sioc:UserAccount,10>
11 <:Bob  a sioc:UserAccount,12>
```

In the following we indicate with t_i the triple at line i. Triples $t_1, \ldots, t_5$ are the TBox, triples t_6 and t_7 are part of the input stream $\mathbb{S}$, while the other triples $(t_8, \ldots, t_{11})$ are the inferred ones. The triples have associated their expiration time: for the TBox triples the expiration time is ∞; triples t_6 and t_7 have expiration time $e = \tau + \omega$ (respectively 10 and 12); triples t_8 and t_{10} inherit their expiration time from t_6 (10); in a similar way the triples t_9 and t_{11} inherit their expiration time from t_7 (12).

At $\tau_2 = 11$, W_{IMaRS} updates its contents (it has slide $\beta = 1$), so the triple t_{12}=`<:Adam sioc:creator_of :tweet3>:[10]` of $\mathbb{S}$ should be added to W_{IMaRS}. t_{12} is annotated with expiration time 17 and it is placed in the Ins context. At this point the maintenance program $\mathcal{M}^+_{ex}$ is executed. We report a possible execution plan in Table 3.6. In each column is reported the content of the contexts during the execution of the plan. In each row a maintenance rule is applied; the triples that are added to contexts by the rules are highlighted in bold. In addition to the triples we described above, in the table there are also t_{13} and t_{14}: triple t_{13} is `<:tweet3 sioc:has_creator :Adam,17>` and it is inferred by t_{12} and t_4 through the rule δ_2^{Ins}; t_{14} is `<:Adam a :UserAccount,17>` and it is inferred by t_{12} and t_5 through the rule δ_1^{Ins}.

TABLE 3.6: Execution of the maintenance program $\mathcal{M}^{+}_{ex}$. Triple derived and added to contexts by the maintenance rule are highlighted in bold.

Rule	Ins	New	Ren	Δ^{-}	Δ^{++}	Δ^{+}
	t_{12}					
δ_1^{New}	t_{12}	$\mathbf{t_1}, \ldots, \mathbf{t_5}$, $\mathbf{t_7}, \mathbf{t_9}, \mathbf{t_{11}}$				
δ_2^{Ins}	t_{12},$\mathbf{t_{13}}$	t_1,…,t_5, t_7,t_9,t_{11}				
δ_1^{Ins}	t_{12},t_{13},$\mathbf{t_{14}}$	t_1,…,t_5, t_7,t_9,t_{11}				
δ_1^{Old}	t_{12},t_{13},t_{14}	t_1,…,t_5, t_7,t_9,t_{11}	$\mathbf{t_{10}}$			
δ_2^{New}	t_{12},t_{13},t_{14}	t_1,…,t_5, t_7,t_9,t_{11}, $\mathbf{t_{12}}, \mathbf{t_{13}}$, $\mathbf{t_{14}}$	t_{10}			
δ_2^{-}	t_{12},t_{13},t_{14}	t_1,…,t_5, t_7,t_9,t_{11}, t_{12},t_{13},t_{14}	t_{10}	$\mathbf{t_{10}}$		
δ_1^{-}	t_{12},t_{13},t_{14}	t_1,…,t_5, t_7,t_9,t_{11}, t_{12},t_{13},t_{14}	t_{10}	$\mathbf{t_6}, \mathbf{t_8}$,$t_{10}$		
δ^{++}	t_{12},t_{13},t_{14}	t_1,…,t_5, t_7,t_9,t_{11}, t_{12},t_{13},t_{14}	t_{10}	t_6,t_8,t_{10}	$\mathbf{t_{12}}, \mathbf{t_{13}}$, $\mathbf{t_{14}}$	
δ^{+}	t_{12},t_{13},t_{14}	t_1,…,t_5, t_7,t_9,t_{11}, t_{12},t_{13},t_{14}	t_{10}	t_6,t_8,t_{10}	t_{12},t_{13},t_{14}	$\mathbf{t_{12}}, \mathbf{t_{13}}$, $\mathbf{t_{14}}$

3.5 Related works

The origin of the approach, which we follow in this work, can be found in incremental maintenance of materialized views in deductive databases [7, 14]. In these works, authors researched on how to generate a persistent view in a deductive databases and how to maintain it incrementally through set of updates. They proved that when the number of modification in the database is under a threshold, the incremental maintenance techniques perform orders of magnitude faster than the whole re-computation of the view.

In the Semantic Web community the most relevant work is [15], where authors proposes an algorithm to incrementally maintain an ontological entailment. Their technique is a declarative variant of the *delete and re-derive* (DRed) algorithm of [14]. The general idea of DRed is a three-steps algoritm:

1. **Overestimate the deletions**: starting from the facts that should be deleted, compute the facts that are deducted by them;
2. **Prune the overstimated deletions**: determine which facts can be rederived by other facts;
3. **Insert the new deducted facts**: derive facts that are consequences of added facts and insert them in the materialization.

The version of DRed proposed by [15] is written in Datalog$^{\neg}$. Table 3.7 shows the list of the Datalog predicates used by DRed (as we explained above, they are similar to the contexts used by IMaRS). The extensions of T^{before} and T^{after} are the materialization before and after the execution of DRed. The goal of the algorithm is the computation of two Datalog predicates, T^{+} and T^{-}, that should be respectively added and removed to the materialization to compute the new one. T^{del}, T^{red}, and T^{ins} are the three predicates used for storing the intermediate results of DRed: in the extension of T^{del} are stored the deletions (step 1); in the extension of T^{red} are stored the overestimated deletions (step 2); finally in the extension of T^{ins} are stored the new derivations (step 3).

Given an ontological language $\mathcal{L}$ expressed as set of inference rules[8], DRed derives a maintenance program. As IMaRS, the maintenance program of DRed is composed by two set of rules. The first set is fixed and it is reported in the first part of Table 3.8. The second set is derived by $\mathcal{L}$ through the rewriting functions reported in the second part of Table 3.8.

IMaRS is inspired by DRed; the main difference is that our algorithm makes the assumption that the deletion can be predicted. It is not valid in general, but it holds for stream reasoning: the window operator allows to determine when RDF triples of the stream are removed.

[8] An inference rule $<?\mathtt{s}_1, ?\mathtt{p}_1, ?\mathtt{o}_1> . \ldots . <?\mathtt{s}_n, ?\mathtt{p}_n, ?\mathtt{o}_n> \Rightarrow <?\mathtt{s}, ?\mathtt{p}, ?\mathtt{o}>$ in Datalog can be represented as $\mathtt{P}(\mathtt{s}, \mathtt{p}, \mathtt{o}) : -\mathtt{P}(\mathtt{s}_1, \mathtt{p}_1, \mathtt{o}_1), \ldots, \mathtt{P}(\mathtt{s}_n, \mathtt{p}_n, \mathtt{o}_n)$

TABLE 3.7: Datalog predicates used by DRed

Name	**Content**
T^{before}	Current materialization
T^{+}	Net insertions required to maitain the materialization
T^{-}	Net deletions required to maitain the materialization
T^{del}	The deletions
T^{ins}	The explicit insertions
T^{red}	The triples marked for deletion which can be alternatively rederived
T^{after}	The materialization after the execution of the maintenance program

TABLE 3.8: Maintenance rules and rewriting functions of DRed

Maintenance rules
$\mathtt{T}^{\mathtt{after}}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{before}}(\mathtt{s},\mathtt{p},\mathtt{o}), \mathtt{not}\ \mathtt{T}^{\mathtt{del}}(\mathtt{s},\mathtt{p},\mathtt{o})$
$\mathtt{T}^{\mathtt{after}}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{red}}(\mathtt{s},\mathtt{p},\mathtt{o})$
$\mathtt{T}^{\mathtt{after}}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{ins}}(\mathtt{s},\mathtt{p},\mathtt{o})$
$\mathtt{T}^{+}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{ins}}(\mathtt{s},\mathtt{p},\mathtt{o}), \mathtt{not}\ \mathtt{T}^{\mathtt{before}}(\mathtt{s},\mathtt{p},\mathtt{o})$
$\mathtt{T}^{-} : -\mathtt{T}^{\mathtt{del}}, \mathtt{not}\ \mathtt{T}^{\mathtt{ins}}, \mathtt{not}\ \mathtt{P}^{\mathtt{red}}$
Rewriting functions for inference rules of $\mathcal{L}$
$\mathtt{T}^{\mathtt{red}}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{before}}(\mathtt{s},\mathtt{p},\mathtt{o}), \mathtt{T}^{\mathtt{after}}(\mathtt{s}_1,\mathtt{p}_1,\mathtt{o}_1), \ldots, \mathtt{T}^{\mathtt{after}}(\mathtt{s}_\mathtt{n},\mathtt{p}_\mathtt{n},\mathtt{o}_\mathtt{n})$
$\{\mathtt{T}^{\mathtt{del}}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{before}}(\mathtt{s}_1,\mathtt{p}_1,\mathtt{o}_1), \ldots, \mathtt{T}^{\mathtt{before}}(\mathtt{s}_{\mathtt{i}-1},\mathtt{p}_{\mathtt{i}-1},\mathtt{o}_{\mathtt{i}-1}),$ $\mathtt{T}^{\mathtt{del}}(\mathtt{s}_\mathtt{i},\mathtt{p}_\mathtt{i},\mathtt{o}_\mathtt{i}), \mathtt{T}^{\mathtt{before}}(\mathtt{s}_{\mathtt{i}+1},\mathtt{p}_{\mathtt{i}+1},\mathtt{o}_{\mathtt{i}+1}), \ldots, \mathtt{T}^{\mathtt{before}}(\mathtt{s}_\mathtt{n},\mathtt{p}_\mathtt{n},\mathtt{o}_\mathtt{n})\}$
$\{\mathtt{T}^{\mathtt{ins}}(\mathtt{s},\mathtt{p},\mathtt{o}) : -\mathtt{T}^{\mathtt{after}}(\mathtt{s}_1,\mathtt{p}_1,\mathtt{o}_1), \ldots, \mathtt{T}^{\mathtt{after}}(\mathtt{s}_{\mathtt{i}-1},\mathtt{p}_{\mathtt{i}-1},\mathtt{o}_{\mathtt{i}-1}),$ $\mathtt{T}^{\mathtt{ins}}(\mathtt{s}_\mathtt{i},\mathtt{p}_\mathtt{i},\mathtt{o}_\mathtt{i}), \mathtt{T}^{\mathtt{after}}(\mathtt{s}_{\mathtt{i}+1},\mathtt{p}_{\mathtt{i}+1},\mathtt{o}_{\mathtt{i}+1}), \ldots, \mathtt{T}^{\mathtt{after}}(\mathtt{s}_\mathtt{n},\mathtt{p}_\mathtt{n},\mathtt{o}_\mathtt{n})\}$

Other approaches to Stream Reasoning are ETALIS [3, 2], Sparkwave [11], Streaming Knowledge Bases [16] and Stream Reasoning via Truth Maintenance Systems [13].

ETALIS [3] is a Complex Event Processing system grounding event processing and stream reasoning in Logic Programming. It is based on event-driven backward chaining rules that realise event-driven inferencing as well as RDFS reasoning. It manages time-based windows using two techniques. On the one hand, it verifies the time window constraints during the incremental event detection, thus it does not generate unnecessary intermediary inferences when time constraints are violated. On the other hand, it periodically prunes expired events by generating system events that look for outdated events, delete them, and trigger the computation of aggregated functions (if present) on the remaining events. ETALIS offers EP-SPARQL [2] to process RDF streams. EP-SPARQL is an extension of SPARQL under RDFS entailment regime which combines graph patterns matching and RDFS reasoning with temporal reasoning by bringing in the ETALIS's temporal operators.

IMaRS and ETALIS are largely incomparable. ETALIS focuses on backward temporal reasoning over RDFS, while IMaRS focuses on forward reasoning on RDFS+. The temporal reasoning is peculiar of ETALIS and it is not present in IMaRS. This restricts the comparison to the continuous query answering task only. The evaluation of IMaRS shows that, in the chosen experimental setting, the continuous query answering task over a materialisation maintained by IMaRS is faster that backward reasoning. However, further investigation is needed to comparatively evaluate the two approaches.

Sparkwave [11] is a solution to perform continuous pattern matching over RDF data streams under RDFS entailment regime. It allows to express temporal constraints in the form of time windows while taking into account RDF schema entailments. It is based on the Rete algorithm [10] which was proposed as a solution for production rule systems, but it offers a general solution for matching multiple patterns against multiple object. The Rete algorithm trades memory for performance by building two memory structures that check the intra and inter-pattern conditions over a set of objects, respectively. Sparkwave adds another memory structure, which computes RDFS entailments, in front of the original two. Under the assumption that the ontology does not change, RDFS can be encoded as rules that are activated by a single triple from the stream. Therefore, each triple from the stream can be treated independently and in a stateless way. This guarantees for high throughput. Moreover, Sparkwave adds time-based window support to Rete in an innovative way. While the state-of-the-art [18] uses a separate thread to prune expired matchings, Sparkwave prunes them after each execution of the algorithm without risking deadlocks and keeping the throughput stable.

Sparkwave is very similar to IMaRS on a conceptual level. It offers an efficient implementation of the IMaRS's maintenance program for RDFS. However, the approach proposed by Sparkwave cannot be extended to RDFS+ (i.e., the ontological language targeted by IMaRS). As stated above, RDFS

can be encoded as rules that are activated by a single triple from the stream, whereas the `owl:transitiveProperty` construct of RDFS+, when encoded as a rule, is activated by multiple triples from the stream. This means that the stateless approach of Sparkwave is no longer sufficient. The IMaRS's maintenance program for RDFS+ cannot be implemented in Sparkwave. Future investigation should comparatively evaluate IMaRS and Sparkwave.

Sparkwave is also difficult to compare with ETALIS, because it does not cover temporal reasoning, but the authors of Sparkwave have temporal reasoning in their future work. They intend to rely on existing works reported in [17, 19], which investigate the integration of temporal reasoning in Rete.

Streaming Knowledge Bases [16] is one of the earliest stream reasoners. It uses TelegraphCQ [8] to efficiently handle data stream, and the Jena rule engine to incrementally materialise the knowledge base. The architecture of Streaming Knowledge Bases is similar to the one of the C-SPARQL Engine. It supports RDFS and the `owl:inverseOf` construct (i.e., rules that are activated by a single triple from the stream), therefore the discussion reported above for Sparkwave also applies to it. Unfortunately, the prototype has never been made available and no comparative evaluation results are available.

IMaRS and all the works above trade expressiveness for performance. They use light-weight ontological languages and time-based windows to optimise the reasoning task and, consequently, to perform continuous reasoning tasks with high throughputs. The authors of [13] take a different perspective; they investigate the possibility to optimise Truth Maintenance Systems so to perform expressive incremental reasoning when the knowledge base is subject to a large amount of random changes (both updates and deletes). They optimise their approach to reason with $\mathcal{EL}$++, the logic underpinning OWL 2 EL, and provide experimental evidence that their approach outperform re-materialisation up to 10% of changes.

3.6 Evaluation

In this section, we report the evaluation of IMaRS. As described in [5], we set up a set of experiments to measure the materialization time. In each experiment there is a window that slides over a stream, and at each update of the window content we measured the time required to compute the materialization. We considered three three different methods:

- **naive**: the materialization is recomputed every time the content of the window changes;
- **DRed-LP**: the materialization is computed through DRed, applying the algorithm described in [15];

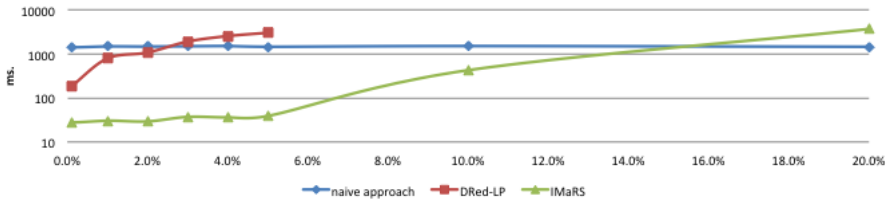

FIGURE 3.4: Evaluation results

- **IMaRS**: the materialization is computed through IMaRS.

DRed-LP and IMaRS have been implemented on the top of the Jena Generic Rule Engine[9].

The input streams are generated by a synthetic data generator. In each experiment we vary the percentage of changes in the window content, i.e., the number of triples in the window and the number of triples that are added and removed when the window slides. The schema used by the input stream is composed by a transitive property `:discuss`; the property relates two messages to indicate a that a message replies another one. For example the RDF triple:

$$<:\texttt{tweet}_i \ :\texttt{discuss} \ :\texttt{tweet}_j>:[\tau]$$

states that $:\texttt{tweet}_j$ is discussed by $:\texttt{tweet}_i$. Even if we consider a simple TBox with one transitive property, the experiment is significant because transitivity is widely used in ontological Web languages (e.g., `rdfs:subClassOf`, `owl:sameAs`, `owl:equivalentClass`). Moreover, transitive properties often generate high numbers of facts, so they stress the system. Finally, the presence of transitivity make the ontological language no first-order rewritable, so we focus on a study-case where the query answer can not be performed through query-rewriting techniques.

The results of the experiments are reported in Figure 3.4. On the x-axis there is the percentage of changes in the window, while in the y-axis there is the time required to compute the new materialization. The naive approach does not depend by the number of changes in the window content, due to the fact that it recomputes the whole materialization every time. As explained above, both the incremental techniques perform better than the naive one when the number of the changes in the window are below certain thresholds.

The threshold for DRed is 2.5%. IMaRS is an order of magnitude faster than DRed for up to 0.1% of changes and continues to be two order faster up to 2.5%. Time performance of IMaRS starts to decrease when the changes are

[9]Cf. `http://jena.apache.org/documentation/inference/index.html#rules`

higher than 8%, and it no longer pays off with respect to the naive approach when the percentage is above 13%.

3.7 Conclusions and future works

In incremental maintenance approaches, developed first for view maintenance in deductive databases and then for other applications, such as the ontological entailment maintenance, the major problem is the deletion: in general, it is not possible to know when a fact would be removed. This is not true in an RDF stream engine: the window operator slides over the input stream with regularity; as consequence, as soon as a triple enters in the scope of the window, IMaRS can compute when it will exit the scope of the window. We exploited this fact to design IMaRS, an incremental maintenance algorithm that extends DRed to maintain the ontological entailment of the content of a window.

When the window has the size parameter higher than the slide parameter the window content changes very often and only little portion of content are added and removed[10]; this is the case where IMaRS performs in the better way. When the window is tumbling (i.e. the stream is partitioned and the window content completely changes every time) the incremental approaches usually fail and the naive one has better time performance.

We analysed a borderline case, where the TBox is static and the whole ABox is dynamic and contained in the data stream. In the general case, the ABox assertions are both dynamic and static: in addition to the data streams, there are static ABox assertions, usually stored in one or more knowledge bases. Additionally, the volume of the static knowledge is much greater than the one of the data stored in the window. In this setting, the IMaRS incremental maintenance approach is faster than the naive one: the percentage of variations at each step is limited (even if the window is tumbling)[11].

At the best of our knowledge, at the moment IMaRS has two implementations. The first is the proof-of-concept we developed at Politecnico di Milano: it is developed over the Jena Generic Rule engine and it uses forward chaining to execute the maintenance program. It is not publicly available; at the moment we are working to implement it in an RDF stream engine. The second (partial) implementation is in Sparkwave [11]: in this work the authors implemented IMaRS's maintenance program for RDFS using an extended version of the Rete algorithm.

We foresee several extensions to this work. An open problem is the multi-

[10] Under the assumption that the elements in the input stream are uniformly distributed.

[11] In a similar way to the TBox assertions, the static ABox assertions do not expire, i.e., they have expiration time ∞

query over a single stream: several queries (each of them with its own window definition) are registered over the same stream. It is very common in stream applications, with a set of windows that slide over the stream at different intervals. A possible solution is to build the "maximal common sub-window" and apply IMaRS over it; this is an original instance of multi-query optimization problem and it is possible when query are pre-registered (such as in stream engines).

Another direction we would like to focus on is related the reasoning part of IMaRS. We want to study how to extend IMaRS to support a language more expressive than RDFS+. First, we would like to investigate the extension to the OWL 2 RL profile: in order to do so we should understand how IMaRS should react when it finds inconsistency in the materialization. Second, it can be interesting to include in IMaRS negation-as-failure. The maintenance program generator should be extended to be able to process rules where the head is *false* (such as in **prp-irp** of OWL 2 RL). In deductive database this problem has been studied by [7].

We are also interested in the problem of relaxing the constraint of the absence of TBox statements in the stream. The main issue is to understand what a TBox axiom with a time stamp τ in a stream means: is the statement valid since τ? Or is it valid also in the past? The answer to these questions influence the way the system should handle the statement and how this statement should be considered in the materialization process. A possible solution could be to consider a temporal extension of the ontological language to cope with this problem.

Bibliography

[1] Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[2] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *WWW*, pages 635–644. ACM, 2011.

[3] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. ETALIS: Rule-Based Reasoning in Event Processing. In Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 99–124. Springer Berlin / Heidelberg, 2011.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[5] Davide Barbieri, Daniele Braga, and Stefano Ceri. Incremental reasoning on streams and rich background knowledge. In *7th Extended Semantic Web Conference, ESWC 2010*, pages 1–15, Heraklion (Greece), 2010. Springer Berlin Heidelberg.

[6] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.

[7] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[8] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A.

Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[9] Emanuele Della Valle, Stefano Ceri, Frank Van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[10] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[11] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In François Bry, Adrian Paschke, Patrick Th. Eugster, Christof Fetzer, and Andreas Behrend, editors, *DEBS*, pages 58–68. ACM, 2012.

[12] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. W3c recommendation, W3C, January 2008.

[13] Yuan Ren and Jeff Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In Craig Macdonald, Iadh Ounis, and Ian Ruthven, editors, *CIKM*, pages 831–836. ACM, 2011.

[14] Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 75–86, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[15] Raphael Volz, Steffen Staab, and Boris Motik. Journal on data semantics ii. chapter Incrementally maintaining materializations of ontologies stored in logic databases, pages 1–34. Springer-Verlag, Berlin, Heidelberg, 2005.

[16] Onkar Walavalkar, Anupam Joshi, Tim Finin, and Yelena Yesha. Streaming Knowledge Bases. In *Proceedings of the Fourth International Workshop on Scalable Semantic Web knowledge Base Systems*, October 2008.

[17] Karen Walzer, Tino Breddin, and Matthias Groch. Relative temporal constraints in the Rete algorithm for complex event detection. In Roberto Baldoni, editor, *DEBS*, volume 332 of *ACM International Conference Proceeding Series*, pages 147–155. ACM, 2008.

[18] Karen Walzer, Matthias Groch, and Tino Breddin. Time to the rescue - supporting temporal reasoning in the rete algorithm for complex event processing. In Sourav S. Bhowmick, Josef Küng, and Roland Wagner, editors, *DEXA*, volume 5181 of *Lecture Notes in Computer Science*, pages 635–642. Springer, 2008.

[19] Karen Walzer, Thomas Heinze, and Anja Klein. Event Lifetime Calculation based on Temporal Relationships. In Jan L. G. Dietz, editor, *KEOD*, pages 269–274. INSTICC Press, 2009.