Incremental Reasoning on Streams and Rich Background Knowledge

Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus

Politecnico di Milano – Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 - 20133 Milano – Italy
{dbarbieri,braga,ceri,dellavalle,grossniklaus}@elet.polimi.it

Abstract. This article presents a technique for Stream Reasoning, consisting in incremental maintenance of materializations of ontological entailments in the presence of streaming information. Previous work, delivered in the context of deductive databases, describes the use of logic programming for the incremental maintenance of such entailments. Our contribution is a new technique that exploits the nature of streaming data in order to efficiently maintain materialized views of RDF triples, which can be used by a reasoner.

By adding expiration time information to each RDF triple, we show that it is possible to compute a new complete and correct materialization whenever a new window of streaming data arrives, by dropping explicit statements and entailments that are no longer valid, and then computing when the RDF triples inserted within the window will expire. We provide experimental evidence that our approach significantly reduces the time required to compute a new materialization at each window change, and opens up for several further optimizations.

1 Introduction

Streaming data is an important class of information sources. Examples of data streams are Web logs, feeds, click streams, sensor data, stock quotations, locations of mobile users, and so on. Streaming data is received continuously and in real-time, either implicitly ordered by arrival time, or explicitly associated with timestamps. A new class of database systems, called data stream management systems (DSMS), is capable of performing queries over streams [1], but such systems cannot perform complex reasoning tasks. Reasoners, on the other hand, can perform complex reasoning tasks, but they do not provide support to manage *rapidly* changing worlds.

Recently, we have made the first steps into a new research direction: Stream Reasoning [2] is a new multi-disciplinary approach that can provide the abstractions, foundations, methods, and tools required to integrate data streams, the Semantic Web, and reasoning systems. Central to the notion of stream reasoning is a paradigmatic change from persistent knowledge bases and user-invoked reasoning tasks to transient streams and continuous reasoning tasks.

© Springer-Verlag Berlin Heidelberg 2010

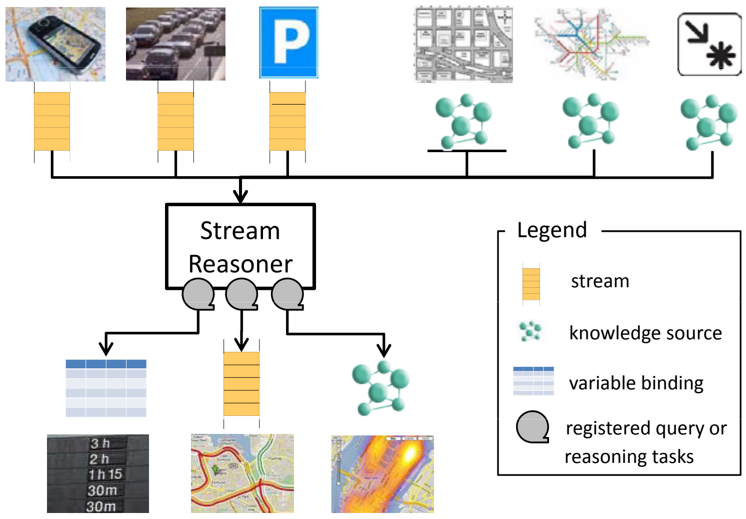

Fig. 1. Mobile Scenario

The first step for enabling Stream Reasoning is the development of languages and systems for querying RDF data also in the form of data streams. Streaming SPARQL [3], Continuous SPARQL (C-SPARQL) [4,5], and Time-Annotated SPARQL [6] are three recent independent proposals for extending SPARQL to handle both static RDF graphs and transient streams of RDF triples. This paper builds on our previous works on C-SPARQL.

In Fig. 1, we show a Stream Reasoner. It takes several streams of rapidly changing information and several static sources of background knowledge as input. In the context of a mobile scenario, examples of sources of streaming data can be the positions of users, the traffic in the streets, and the availability of parking lots, whereas examples of background knowledge can be the city layout, the public transportation schedules, and the descriptions of points of interest and of events in a given area. Several reasoning tasks, expressed in the form of C-SPARQL queries, are registered into the stream reasoner, and the system continuously generates new answers. These answers can be in the standard SPARQL output form (i.e., variable bindings and graphs) or in the form of streams. In our mobile scenario, for instance, we can register two C-SPARQL queries: one continuously monitors the status of the public transportation system and returns the delays as variable bindings, the other one monitors the sensors for traffic detection and generates a stream of aggregate information for each major road. Current implementations of the proposed SPARQL extensions, however, assume only a simple entailment (see Section 2 of [7]). They do not try to handle reasoning on streaming information, e.g., providing strategical suggestions about how to perform goals.

In existing work on logical reasoning, the knowledge base is always assumed to be static (or slowly evolving). There is work on changing beliefs on the basis of new observations [8], but the solutions proposed in this area are far too complex to be applicable to gigantic data streams of the kind we image in a mobile context. However, the nature of data streams is different from arbitrary changes, because change occurs in a "regular" way at the points where the streaming data is observed.

In this article, we present a technique for stream reasoning that incrementally maintains a materialization of ontological entailments in the presence of streaming information. We elaborate on previous papers [9,10] that extend to logic programming results from incremental maintenance of materialized views in deductive databases [11]. Our contribution is a new technique that takes the order in which streaming information arrives at the Stream Reasoner into explicit consideration. By adding expiration time information to each RDF statement, we show that it is possible to compute a new complete and correct materialization by (a) dropping explicit statements and entailments that are no longer valid, and (b) evaluating a maintenance program that propagates insertions of explicit RDF statements as changes to the stored implicit entailments.

The rest of the paper is organized as follows. Section 2 presents a wrap up of the background information needed to understand this paper. In particular, it presents the state of the art in incremental maintenance of materializations of ontologies represented as logic programs. Section 3 presents our major contribution in the form of Datalog rules computing the incremental materialization of ontologies for window-based changes of ontological entailments. In Section 4 we present our implementation experience. Section 5 provides experimental evidence that our approach significantly reduces the time required to compute the new materialization. Finally, we close the paper by sketching future works in Section 6.

2 Background

2.1 Stream Reasoning

A first step toward stream reasoning has been to combine the power of existing data-stream management systems and the Semantic Web [12]. The key idea is to keep streaming data in relational format as long as possible and to bring it to the semantic level as aggregated events [5]. Existing data models, access protocols, and query languages for data-stream management systems and the Semantic Web are not sufficient to do so and, thus, they must be combined.

C-SPARQL [4,5] introduces the notion of RDF streams as the natural extension of the RDF data model to this scenario, and then extend SPARQL to query RDF streams. An RDF stream is defined as an ordered sequence of pairs, where each pair is constituted by an RDF triple and its timestamp τ.

$$\begin{gathered} \dots \\ (\langle subj_i, pred_i, obj_i \rangle \ , \ \tau_i) \\ (\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle \ , \ \tau_{i+1}) \\ \dots \end{gathered}$$

Fig. 2 shows an example of a C-SPARQL query that continuously queries a RDF stream as well as a static RDF graph. The RDF stream describes the users sitting in trains and trains moving from a station to another one. The RDF graph describes where the stations are located, e.g., a station is in a city, which is in a region.

```
 1. REGISTER QUERY TotalAmountPerBroker COMPUTE EVERY 1sec AS
 2. PREFIX ex: <http://example/>
 3. SELECT DISTINCT ?user ?type ?x
 4. FROM <http://mobileservice.org/meansOfTransportation.rdf>
 5. FROM STREAM <http://mobileservice.org/positions.trdf>
 5. [RANGE 10sec STEP 1sec]
 6. WHERE {
 7.    ?user ex:isIn ?x .
 8.    ?user a ex:Commuter .
 9.    ?x a ?type .
10.    ?user ex:remainingTravelTime ?t .
11.    FILTER (?t >= "PT30M"xsd:duration )
12. }
```

Fig. 2. An example of C-SPARQL query that continuously queries a RDF stream as well as a static RDF graph

At line 1, the REGISTER clause instructs the C-SPARQL engine to register a continuous query. The COMPUTE EVERY clause states the frequency of every new computation. In line 5, the FROM STREAM clause defines the RDF stream of positions used in the query. Next, line 6 defines the window of observation of the RDF stream. Streams, by their very nature, are volatile and consumed on the fly. The C-SPARQL engine, therefore, observes them through a window that contains the stream's most recent elements and that changes over time. In the example, the window comprises RDF triples produced in the last 10 seconds and the window slides every second. The WHERE clause is standard SPARQL as it includes a set of matching patterns, which restricts users to be commuters and a FILTER clause, which restricts the answers to users whose remaining traveling time is at least 30 minutes. This example shows that, at the time of the presentation in the window, it is possible to compute the time when triples both of the window and of ontological entailments will cease to be valid.

2.2 Expressing Ontology Languages as Rules

Using rules is a best practice (see Section 2.1 of [9]) in implementing the logical entailment supported by ontology languages such as RDF-S [13] and OWL2-RL [14]. For example, Fig. 3 presents the set of rule used by the Jena Generic Rule Engine [15] to compute RDF-S closure. The first rule (rdfs2) states that if there is a triple `<?x ?p ?y>` and the domain of the property `?p` is the class

```
[rdfs2: (?x ?p ?y), (?p rdfs:domain ?c) -> (?x rdf:type ?c)]
[rdfs3: (?x ?p ?y), (?p rdfs:range ?c) -> (?y rdf:type ?c)]
[rdfs5a: (?a rdfs:subPropertyOf ?b), (?b rdfs:subPropertyOf ?c)
          -> (?a rdfs:subPropertyOf ?c)]
[rdfs5b: (?a rdf:type rdf:Property) -> (?a rdfs:subPropertyOf ?a)]
[rdfs6: (?a ?p ?b), (?p rdfs:subPropertyOf ?q) -> (?a ?q ?b)]
[rdfs7: (?a rdf:type rdfs:Class) -> (?a rdfs:subClassOf ?a)]
[rdfs8: (?a rdfs:subClassOf ?b), (?b rdfs:subClassOf ?c)
          -> (?a rdfs:subClassOf ?c)]
[rdfs9: (?x rdfs:subClassOf ?y), (?a rdf:type ?x) -> (?a rdf:type ?y)]
[rdfs10: (?x rdf:type rdfs:ContainerMembershipProperty)
          -> (?x rdfs:subPropertyOf rdfs:member)]
[rdf1and4: (?x ?p ?y) -> (?p rdf:type rdf:Property),
                         (?x rdf:type rdfs:Resource),
                         (?y rdf:type rdfs:Resource)]
[rdfs7b: (?a rdf:type rdfs:Class) -> (?a rdfs:subClassOf rdfs:Resource)]
```

Fig. 3. Rules Implementing RDF-S in Jena Generic Rule Engine

`?c` (represented by the triple `<?p rdfs:domain ?c>`) then the resource `?x` is of type `?c` (represented by the triple `<?x rdf:type ?c>`).

In the rest of the paper, we adopt logic programming terminology. We refer to a set of rules as a *logic program* (or simply program) and we assume that any RDF graph can be stored in the extension of a single ternary predicate P. Under this assumption, the rule rdfs2 can be represented in Datalog as follows.

$$P(x, rdf:type, c) \text{:-} P(p, rdfs:domain, C), P(s, p, y)$$

2.3 Incremental Maintenance of Materializations

Maintenance of a materialization when facts change, i.e., facts are added or removed from the knowledge base, is a well studied problem. The state of the art approach implemented in systems such as KAON[1] is a declarative variant [9] of the delete and re-derive (DRed) algorithm proposed in [16]. DRed incrementally maintains a materialization in three steps.

1. Overestimate the deletions by computing all the direct consequences of a deletion.
2. Prune the overestimated deletions for which the deleted fact can be re-derived from other facts.
3. Insert all derivation which are consequences of added facts.

More formally, a logic program is composed by a set of rules $\mathbf{R}$ that we can represent as H :- $B_1, \ldots, B_n$, where H is the predicate that forms the head of the rule and $B_1, \ldots, B_n$ are the predicates that form the body of the rule. If we

[1] The Datalog engine is part of the KAON suite, see `http://kaon.semanticweb.org`

call the set of predicates in a logic program $\mathbf{P}$, then we can formally assert that $H, B_i \in \mathbf{P}$. A *maintenance program*, which implements the declarative version of the DRed algorithm, can be automatically derived from the original program with a fixed set of rewriting functions (see Table 2) that uses seven maintenance predicates (see Table 1) [9].

Table 1. The maintenance predicates (derived from [9])

Name	Content of the extension
P	the current materialization
P^{Del}	the deletions
P^{Ins}	the explicit insertion
P^{Red}	the triples marked for deletion which have alternative derivations
P^{New}	the materialization after the execution of the maintenance program
P^{+}	the net insertions required to maintain the materialization
P^{-}	the net deletions required to maintain the materialization

Given a materialized predicate P and the set of extensional insertions P^{Ins} to and deletions P^{Dels} from P, the goal of the rewriting functions is the definition of two maintenance predicates P^{+} and P^{-}, such that the extensions of P^{+} and P^{-} contain the net insertions and deletions, respectively, that are needed to incrementally maintain the materialization of P.

Table 2. Rewriting functions (derived from [9])

Predicate		
Name	Generator Parameter	Rewriting Result
δ_1^{New}	$P \in \mathbf{P}$	P^{New} :- $P, not\, P^{Del}$
δ_2^{New}	$P \in \mathbf{P}$	P^{New} :- P^{Red}
δ_3^{New}	$P \in \mathbf{P}$	P^{New} :- P^{Ins}
δ^{+}	$P \in \mathbf{P}$	P^{+} :- $P^{Ins}, notP$
δ^{-}	$P \in \mathbf{P}$	P^{-} :- $P^{Del}, notP^{Ins}, not\, P^{Red}$

Rule		
Name	Generator Parameter	Rewriting Result
δ^{Red}	H :- $B_1, \ldots, B_n$	H^{Red} :- $H^{Del}, B_1^{New}, \ldots, B_n^{New}$
δ^{Del}	H :- $B_1, \ldots, B_n$	$\{H^{Del}$:- $B_1, \ldots, B_{i-1}, B_i^{Del}, B_{i+1}, \ldots, B_n\}$
δ^{Ins}	H :- $B_1, \ldots, B_n$	$\{H^{Ins}$:- $B_1^{New}, \ldots, B_{i-1}^{New}, B_i^{Ins}, B_{i+1}^{New}, \ldots, B_n^{New}\}$

We can divide the rewriting functions shown in Table 2 in two groups. One group of functions apply to predicates, while the other group of functions apply to rules. The former functions use the predicates defined in Table 1 to introduce the rules that will store the materialization after the execution of the maintenance program in the extension of the predicate P^{New}. The latter functions introduce the rules that populate the extensions of the predicates P^{Del}, P^{Red}, and P^{Ins}.

These three rewriting functions are executed for each rule that has the predicate P as head. While the function δ^{Red} rewrites each rule in exactly one maintenance rule, the two functions δ^{Del} and δ^{Ins} rewrite each rule with n bodies B_i into n maintenance rules.

To exemplify how these rewriting functions work in practice, let us return to the scenario exemplified in Sect. 2.1. To describe that scenario, we introduced the predicate *isIn* that captures the respective position of moving objects (e.g., somebody is in a train, the train is in station, somebody else is in a car, the car is in a parking lot, etc.). A simple ontology for a mobility scenario could express transitivity and be represented using the following Datalog rule.

$$(R)\ isIn(x, z) \text{ :- } isIn(x, y), isIn(y, z)$$

By applying the rewriting functions presented in Table 2 to the rule (R) and the predicate *isIn*, we obtain the maintenance program shown in Table 3. Each row of the table contains the applied rewriting function and the rewritten maintenance rule.

Table 3. The maintenance program automatically derived from a program containing only the rule R by applying the rewriting functions show in Table 2

Rule	Rewriting Function
$isIn^{New}(x,y) \text{ :- } isIn(x,y), not\, isIn^{Del}(x,y)$	$\delta_1^{New}(isIn)$
$isIn^{New}(x,y) \text{ :- } isIn^{Red}(x,y)$	$\delta_2^{New}(isIn)$
$isIn^{New}(x,y) \text{ :- } isIn^{Ins}(x,y)$	$\delta_3^{New}(isIn)$
$isIn^{+}(x,y) \text{ :- } isIn^{Ins}(x,y), not\, isIn(x,y)$	$\delta^{+}(isIn)$
$isIn^{-}(x,y) \text{ :- } isIn^{Del}(x,y), not\, isIn^{Ins}(x,y), not\, isIn^{Red}(x,y)$	$\delta^{-}(isIn)$
$isIn^{Red}(x,z) \text{ :- } isIn^{Del}(x,z), isIn^{New}(x,y), isIn^{New}(y,z)$	$\delta^{Red}(R)$
$isIn^{Del}(x,z) \text{ :- } isIn^{Del}(x,y), isIn(y,z)$	$\delta^{Del}(R)$
$isIn^{Del}(x,z) \text{ :- } isIn(x,y), isIn^{Del}(y,z)$	$\delta^{Del}(R)$
$isIn^{Ins}(x,z) \text{ :- } isIn^{Ins}(x,y), isIn^{new}(y,z)$	$\delta^{Ins}(R)$
$isIn^{Ins}(x,z) \text{ :- } isIn^{new}(x,y), isIn^{Ins}(y,z)$	$\delta^{Ins}(R)$

3 Maintaining Materialization of RDF Streams

As we explained earlier in this paper, incremental maintenance of materializations of ontological entailments after knowledge changes is a well studied problem. However, additions or removals of facts from the knowledge base induced by data streams are governed by windows, which have a known expiration time. The intuition behind our approach is straightforward. If we tag each RDF triple (both explicitly inserted and entailed) with a *expiration time* that represents the last moment in which it will be in the window, we can compute a new complete and correct materialization by dropping RDF triples that are no longer in the window and then evaluate a maintenance program that

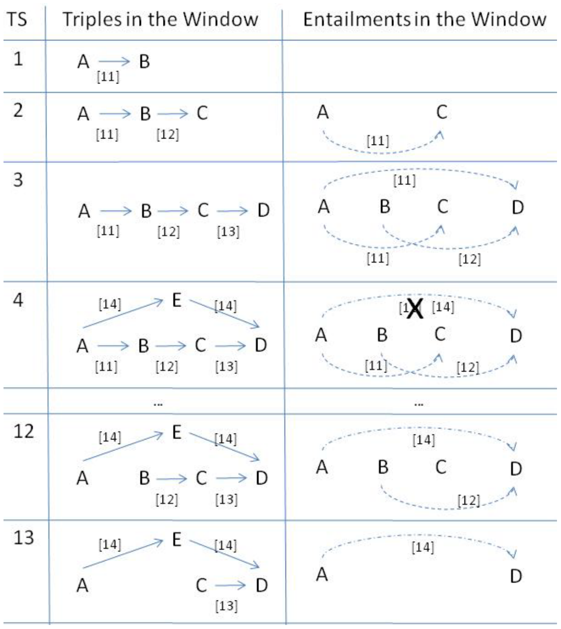

Fig. 4. Our approach to incrementally maintain the materialization at work

1. computes the entailments derived by the inserts,
2. annotates each entailed triple with a expiration time, and
3. eliminates from the current state all copies of derived triples except the one with the highest timestamp.

Note that this approach supports the immediate deletions of both window facts and entailed triples which are dropped by inspection to their expiration times. Instead it requires some extra work for managing insertions as new timestamps need to be computed. This approach is more effective than overestimating the deletions and then computing re-derivations, as we will demonstrate in Section 5.

Figure 4 illustrates our approach. Let us assume that we have a stream of triples in which all the triples use the same predicate *isIn* introduced in Section 2.3. Let us also assume that we register a simple C-SPARQL query that observes an RDF stream through a sliding window of 10 seconds and computes the transitive closure of the *isIn* property.

In the 1st second of execution, the triple `<A isIn B>` enters the window. We tag the triple with the expiration time 11 (i.e., it will be valid until the 11th second) and no derivation occurs. The transitive closure only contains that triple. In the 2nd second the triple `<B isIn C>` enters the window. We can tag it with the expiration time 12 and we can materialize the entailed triple `<A isIn C>`. As the triple `<A isIn B>` expires in the 11th second, the entailed triple `<A isIn C>` also expires then and, thus, we tag it with the expiration time 11 (i.e., Step 2 of our approach). As the 11th second passes, we will have to just drop the

Table 4. The maintenance predicates of our approach

Name	Content of the extension
P	the current materialization
P^{Ins}	the triples that enter the window
P^{New}	the triples which are progressively added to the materialization
P^{Old}	the triples for which re-derivations with a longer expiration time were materialized
P^{+}	the net insertions required to maintain the materialization
P^{-}	the net deletions required to maintain the materialization

triples tagged with 11 and the materialization will be up to date (i.e., Step 1 of our approach).

Let us then assume that in the 3rd second, the triple `<C isIn D>` enters the window. We tag it with the expiration time 13 and compute two entailments: the triple `<B isIn D>` with expiration time 12 and the triple `<A isIn D>` with expiration time 11. In the 4th second, the two triples `<A isIn E>` and `<E isIn D>` enter the window. Both triples are tagged with the expiration time 14. We also derive the entailed triple `<A isIn D>` with time expiration 14. The triple `<A isIn D>` was previously derived, but its expiration time was 11 and, therefore, that triple is dropped. The rest of Fig. 4 shows how triples are deleted when they expire.

More formally, our logic program is composed of a set of rules $\mathbf{R}$ that we can represent as $H[T]$:- $B_1[T_1], \dots, B_n[T_n]$, where H is the predicate that forms the head of the rule and it is valid until T. $B_1[T_1], \dots, B_n[T_n]$ are the n predicates that form the body of the rule with their respective n expiration times $T_1 \dots T_n$. As in the case illustrated in Section 2.3, we can formally assert that $H, B_i \in \mathbf{P}$ where $\mathbf{P}$ denotes the set of predicates in a logic program.

Table 5. The rewriting functions of our approach

Predicate		
Name	Generator Parameter	Rewriting Result
Δ_1^{New}	$P \in \mathbf{P}$	$P^{New}[T]$:- $P[T], not P[T_1], T_1 = (now - 1)$
Δ_2^{New}	$P \in \mathbf{P}$	$P^{New}[T]$:- $P^{Ins}[T], not\, P^{Old}[T]$
Δ_1^{Old}	$P \in \mathbf{P}$	$P^{Old}[T]$:- $P^{Ins}[T_1], P[T], T1 > T$
Δ_2^{Old}	$P \in \mathbf{P}$	$P^{Old}[T]$:- $P^{Ins}[T_1], P^{Ins}[T], T1 > T$
Δ_1^{-}	$P \in \mathbf{P}$	$P^{-}[T]$:- $P[T_1], T_1 = (now - 1), not\, P^{Ins}[T_1]$
Δ_2^{-}	$P \in \mathbf{P}$	$P^{-}[T]$:- $P^{Old}[T]$
Δ^{++}	$P \in \mathbf{P}$	$P^{++}[T]$:- $P^{New}[T], not\, P[T_1]$
Δ^{+}	$P \in \mathbf{P}$	$P^{+}[T]$:- $P^{++}[T], not\, P^{Old}[T_1]$

Rule		
Name	Generator Parameter	Rewriting Result
Δ^{Ins}	H :- $B_1, \dots, B_n$	$\{H^{Ins}[T]$:- $B_1^{New}[T_1], \dots, B_{i-1}^{New}[T_{i-1}]$, $B_i^{Ins}[T_i], B_{i+1}^{New}[T_{i+1}], \dots, B_n^{New}[T_n]$, $T = min(T_1, \dots, T_n)\}$

A maintenance program, which implements our approach in a declarative way, can be automatically be derived from the original program with a fixed set of rewriting functions (see Table 5) that uses five maintenance predicates (see Table 4) inspired by the approach of Volz et al. [9].

Given a materialized predicate P and set of extensional insertions P^{Ins} determined by the new triple entering the window, the goal of the rewriting functions is the definition of the maintenance predicates P^{+} and P^{-} whose extension contains the net insertions and the net deletions needed to incrementally maintain the materialization of P. The extension of the maintenance predicate P^{-} contains the extensions of predicate P that expires as well as the extension of predicate P^{Old}. In Table 5 we formally defines our rewriting functions. Note that P^{++} is only an auxiliary predicate with not special meaning.

By applying the rewriting functions presented in Table 5 to the rule (R) and the predicate *isIn* defined in Section 2.3, we obtain the maintenance program shown in Table 6.

Table 6. The maintenance program automatically derived from a program containing only the rule R by applying the rewriting functions show in Table 5

Rule	Function
$isIn^{New}(x,y)[T]$:- $isIn(x,y)[T], not\, isIn(x,y)[T_1], T_1 = (now - 1)$	$\Delta_1^{New}(isIn)$
$isIn^{New}(x,y)[T]$:- $isIn^{Ins}(x,y)[T], notisIn^{Old}(x,y)[T]$	$\Delta_2^{New}(isIn)$
$isIn^{Old}(x,y)[T]$:- $isIn^{Ins}(x,y)[T_1], isIn(x,y)[T], T1 > T$	$\Delta_1^{Old}(isIn)$
$isIn^{Old}(x,y)[T]$:- $isIn^{Ins}(x,y)[T_1], isIn^{Ins}(x,y)[T], T1 > T$	$\Delta_2^{Old}(isIn)$
$isIn^{-}(x,y)[T]$:- $isIn(x,y)[T_1], T_1 = (now - 1), not\, isIn^{Ins}(x,y)[T_1]$	$\Delta_1^{-}(isIn)$
$isIn^{-}(x,y)[T]$:- $isIn^{Old}(x,y)[T]$	$\Delta_2^{-}(isIn)$
$isIn^{++}(x,y)[T]$:- $isIn^{New}(x,y)[T], not\, isIn(x,y)[T_1]$	$\Delta^{++}(isIn)$
$isIn^{+}(x,y)[T]$:- $isIn^{++}(x,y)[T], not\, isIn^{Old}(x,y)[T_1]$	$\Delta^{+}(isIn)$
$isIn^{Ins}(x,z)[T]$:- $isIn^{Ins}(x,y)[T_1], isIn^{New}(y,z)[T_2], T = min(T_1,T_2)$	$\Delta^{Ins}(R)$
$isIn^{Ins}(x,z)[T]$:- $isIn^{New}(x,y)[T_1], isIn^{Ins}(y,z)[T_2], T = min(T_1,T_2)$	$\Delta^{Ins}(R)$

4 Implementation Experience

Figure 5 illustrates the architecture of our current prototype, implemented by using the Jena Generic Rule Engine. The *Incremental Maintainer* component orchestrates the maintenance process. It keeps the current materialization in the *Permanent Space* and uses the *Working Space* to compute the net inserts and deletes. Both spaces consist of an RDF store for the triples and a hashtable which caters for efficient management of the expiration time associated with each triple.

The maintenance program (see Fig. 6) is loaded into the rule engine that operates over the RDF store in the working space. The management of expiration times is performed by using four custom built-ins, *GetVT*, *GetDiffVT*, *SetVT* and *DelVT*, that are triggered by the maintenance program[2]. GetVT retrieves

[2] For more information on how to write built-ins for Jena Generic Rule Engine see [15].

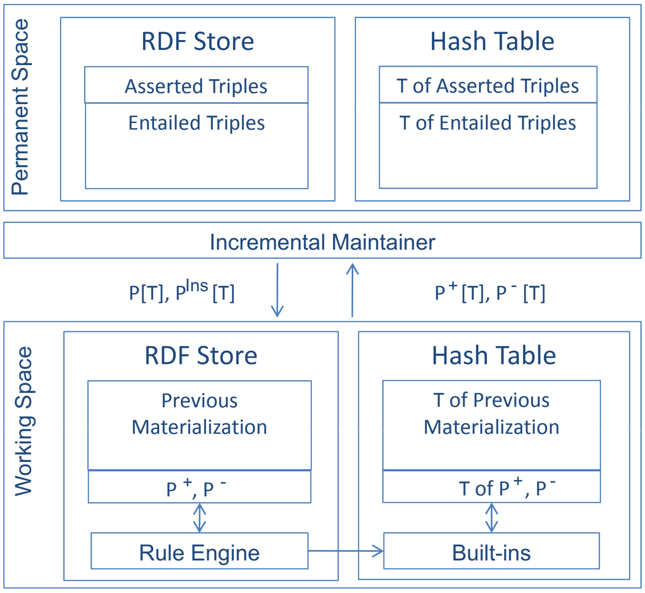

Fig. 5. Overview of the prototype implementation

```
[New1: (?A isIn ?B), GetVT(?A isIn ?B, ?T), noValue(?A isInExp ?B)
        -> (?A isInNew ?B), SetVT(?A isInNew ?B, ?T)]
[New2: (?A isInIns ?B), GetVT(?A isInIns ?B, ?T), noValue(?A isInOld ?B)
        -> (?A isInNew ?B), SetVT(?A isInNew ?B, ?T)]
[Old1: (?A isInIns ?B), GetVT(?A isInIns ?B, ?T1),
       (?A isIn ?B), GetVT(?A isIn ?B, ?T), lessThan(?T, ?T1)
        -> (?A isInOld ?B), DelVT(?A isInIns ?B, ?T)]
[Old2: (?A isInIns ?B), GetVT(?A isInIns ?B, ?T1),
       (?A isInIns ?B), GetDiffVT(?A isIn ?B, ?T1, ?T), lessThan(?T, ?T1)
        -> (?A isInOld ?B), DelVT(?A isInIns ?B, ?T)]
[Rem1: (?A isInExp ?B), GetVT(?A isInExp ?B, ?T), noValue(?A isInIns ?B)
        -> (?A isInRem ?B), DelVT(?A isInExp ?B, ?T) ]
[Rem2: (?A isInOld ?B) -> (?A isInRem ?B) ]
[Add2: (?A isInNew ?B), GetVT(?A isInNew ?B, ?T), noValue(?A isIn ?B)
        -> (?A isInAdd2 ?B), SetVT(?A isInAdd2 ?B, ?T) ]
[Add1: (?A isInAdd2 ?B), GetVT(?A isInAdd2 ?B, ?T), noValue(?A isInOld ?B)
        -> (?A isInAdd ?B), SetVT(?A isInAdd ?B, ?T) ]
[Ins1: (?A isInIns ?B), GetVT(?A isInIns ?B, ?T1),
       (?B isInNew ?C), GetVT(?B isInNew ?C, ?T2), min(?T1, ?T2, ?T)
        -> (?A isInIns ?C), SetVT(?A isInIns ?C, ?T)]
[Ins2: (?A isInNew ?B), GetVT(?A isInNew ?B, ?T1),
       (?B isInIns ?C), GetVT(?B isInIns ?C, ?T2), min(?T1, ?T2, ?T)
        -> (?A isInIns ?C), SetVT(?A isInIns ?C, ?T)]
```

Fig. 6. The maintenance program shown in Table 6 implemented in Jena Generic Rule Engine

the expiration time of a triple from the hashtable; GetDiffVT gets possible other expiration times of a given triple and is used to efficiently implement the rules generated by Δ_2^{Old}; SetVT sets the expiration time of a triple in the hashtable; DelVT deletes the expiration time of a triple from the hashtable.

The maintenance process is carried out as follows. When the system is started up, the background knowledge is loaded into the permanent space. Then, the maintenance program is evaluated on the background knowledge and the extension of all predicates P is stored in the RDF store. The expiration time of all triples is set to a default value which indicates that they cannot expire. As the window slides over the stream(s), the incremental maintainer:

(a) puts all triples entering the window in the extension of P^{Ins},
(b) loads the current materialization and P^{Ins} in the working space,
(c) copies the expiration times from the permanent space into the working space,
(d) evaluates the maintenance program,
(e) updates the RDF store in the permanent space by adding the extension of P^+ and removing the extension of P^-,
(f) updates the hash tables by changing the expiration time of the triples in the extension of P^+ and removing from the table the triples of P^-, and
(g) clears the working space for a new evaluation.

5 Evaluation

This section reports on the evaluation we carried out using various synthetically generated data sets that use the transitive property *isIn*. Although we limit our experiments to the transitive property, the test is significant because widely used vocabularies in Web ontological languages are transitive (e.g., rdfs:subClassOf, rdfs:subPropertyOf, owl:sameAs, owl:equivalentProperty, owl:equivalentClass and all properties of type owl:TransitiveProperty). Moreover, transitive properties are quite generative in terms of entailments and, thus, stress the system.

Our synthetic data generator generates trees of triples all using *isIn* as property. We can control the depth of the tree and the number of trees generates. All generated triples are stored in a pool. An experiment consists of measuring the time needed to compute a new materialization based on the given the background knowledge, the triples in the window as well as the triples that enter and exit the window at each step. When we start an experiment, we first extract a subset of triples from the pool to form the background knowledge. Then, we stream the rest of the triples from the pool. We control both the dimension of the window over the stream of triples and the number of triples entering and exiting the window at each step.

In our experiments we compare three approaches: (a) the *naive* approach of recomputing the entire materialization at each step, (b) the maintenance program shown in Table 3 implementing [9], denoted as *incremental-volz*), and (c) our maintenance program shown in Tables 6 and in Fig. 6, denoted as *incremental-stream*. Intuitively, the naive approach is dominated with a small number of

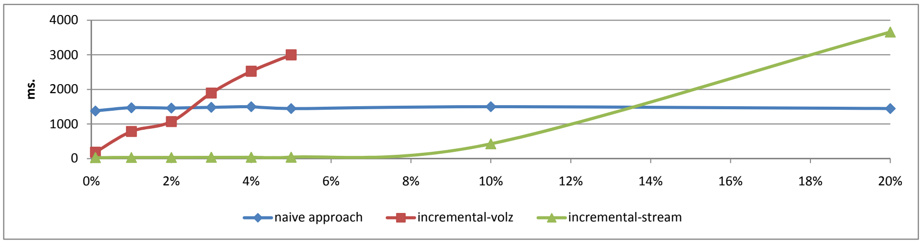

Fig. 7. Evaluation results: the time (ms) required to maintain the materialization as a function of the percentage of the background knowledge subject to change

streaming triples, and dominates when streaming triples are a large fraction of the materialization.

We run multiple experiments[3] using different settings of the experimental environment, by changing the size of the background knowledge, the size of the window, and the number of triples entering and exiting the window at each step. In Fig. 7, we plot the results of one of these experiments (which qualitatively are very similar). We compare the materialization maintenance time as a function of the percentage of the background knowledge subject to change. As one can read from the graph, the incremental-volz [9] approach is faster that the naive approach only if the changes induced by the streaming triples encompass less that 2.5% of the background knowledge. Our incremental-stream approach is an order of magnitude faster than incremental-volz for up to 0.1% of changes and continues to be two orders of magnitude faster up to 2.5% of changes. It no longer pays off with respect to the naive approach when the percentage of change is above 13%.

6 Conclusion and Future Work

In this paper, we have shown how previous work from the field of deductive databases can be applied to the maintenance of ontological entailments with data streams. Our approach is an extension of the algorithm developed by Volz et al. [9], that uses logic programming to maintain materializations incrementally. Data streams use the notion of windows to extract snapshots from streams, that are then processed by the query evaluator; we leverage this fact to define the triples that are inserted into and deleted from the materialization. We have also presented an implementation as an extension of the Jena Generic Rule Engine; our implementation uses hash tables to manage triple expiration time. We have shown that our approach outperforms existing approaches when the window size is a fraction (below 10%) of the knowledge base: this assumption holds for all known data stream applications.

We foresee several extensions to this work. With our approach, at insertion time we explicitly remove old triples which have multiple derivations, but we are

[3] We run all experiment on a Intel® Core^{TM}Duo 2.20 GHz with 2 GB of RAM.

considering the option of keeping all derivations and simply let them expire when they expire, thus simplifying also insertions. Of course, this requires programs (e.g., our C-SPARQL engine) to be aware of the existence of multiple instances of the same triple, with different expiration times, and ignore all but one of such instances. Another open problem is the application of our approach to several queries over the same streams, with several windows that move at different intervals. A possible solution to this problem is to build the notion of "maximal common sub-window" and then apply the proposed algorithm to them. This is an original instance of multi-query optimization, that is indeed possible when queries are preregistered (as with stream databases and C-SPARQL). Finally, we intend to explore a "lazy" approach to materialization, in which only entailments that are needed to answer registered queries are computed. In our future work, we plan to address these issues.

Acknowledgements

The work described in this paper has been partially supported by the European project LarKC (FP7-215535). Michael Grossniklaus's work is carried out under SNF grant number PBEZ2-121230.

References

1. Garofalakis, M., Gehrke, J., Rastogi, R.: Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications). Springer, Heidelberg (2007)
2. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a Streaming World! Reasoning upon Rapidly Changing Information. IEEE Intelligent Systems 24(6), 83–89 (2009)
3. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL - extending SPARQL to process data streams. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 448–462. Springer, Heidelberg (2008)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for Continuous Querying. In: Proc. Intl. Conf. on World Wide Web (WWW), pp. 1061–1062 (2009)
5. Barbieri, D.F., Braga, D., Ceri, S., Grossniklaus, M.: An Execution Environment for C-SPARQL Queries. In: Proc. Intl. Conf. on Extending Database Technology, EDBT (2010)
6. Rodriguez, A., McGrath, R., Liu, Y., Myers, J.: Semantic Management of Streaming Data. In: Proc. Intl. Workshop on Semantic Sensor Networks, SSN (2009)
7. McBride, B., Hayes, P.: RDF Semantics. W3C Recommendation (2004), `http://www.w3.org/TR/rdf-mt/`
8. Gaerdenfors, P. (ed.): Belief Revision. Cambridge University Press, Cambridge (2003)
9. Volz, R., Staab, S., Motik, B.: Incrementally maintaining materializations of ontologies stored in logic databases. J. Data Semantics 2, 1–34 (2005)
10. Staudt, M., Jarke, M.: Incremental maintenance of externally materialized views. In: Vijayaraman, T.M., Buchmann, A.P., Mohan, C., Sarda, N.L. (eds.) VLDB, pp. 75–86. Morgan Kaufmann, San Francisco (1996)

11. Ceri, S., Widom, J.: Deriving production rules for incremental view maintenance. In: Lohman, G.M., Sernadas, A., Camps, R. (eds.) VLDB, pp. 577–589. Morgan Kaufmann, San Francisco (1991)
12. Della Valle, E., Ceri, S., Barbieri, D.F., Braga, D., Campi, A.: A First Step Towards Stream Reasoning. In: Proc. Future Internet Symposium (FIS), pp. 72–81 (2008)
13. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (2004), `http://www.w3.org/TR/rdf-schema/`
14. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: Owl 2 web ontology language: Profiles. W3C Recommendation (2009), `http://www.w3.org/TR/owl2-profiles/`
15. Reynolds, D.: Jena 2 inference support (2009), `http://jena.sourceforge.net/inference/`
16. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: Buneman, P., Jajodia, S. (eds.) SIGMOD Conference, pp. 157–166. ACM Press, New York (1993)