C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing

Pieter Bonte
Ghent University - imec, Belgium
pieter.bonte@ugent.be

Riccardo Tommasini
Politecnico di Milano, DEIB, Italy
riccardo.tommasini@polimi.it

Filip De Turck
Ghent University - imec, Belgium
filip.deturck@ugent.be

Femke Ongenae
Ghent University - imec, Belgium
femke.ongenae@ugent.be

Emanuele Della Valle
Politecnico di Milano, DEIB, Italy
emanuele.dellavalle@polimi.it

ABSTRACT

Many domains, such as the Internet of Things and Social Media, demand to combine data streams with background knowledge to enable meaningful analysis in real-time. When background knowledge takes the form of taxonomies and class hierarchies, Semantic Web technologies are valuable tools and their extension to data streams, namely RDF Stream processing (RSP), offers the opportunity to integrate the background knowledge with RDF streams. In particular, RSP Engines can continuously answer SPARQL queries while performing reasoning. However, current RSP engines are at risk of failing to perform reasoning at the required throughput. In this paper, we formalize continuous hierarchical reasoning. We propose an optimized algorithm, namely C-Sprite, that operates in constant time and scales linearly in the number of continuous queries (to be evaluated in parallel). We present two implementations of C-Sprite: one exploits a language feature often found in existing Stream Processing engines while the other is an optimized implementation. The empirical evaluation shows that the proposed solution is at least twice as fast as current approaches.

KEYWORDS

Stream Processing, RSP, Hierarchical Reasoning

ACM Reference Format:
Pieter Bonte, Riccardo Tommasini, Filip De Turck, Femke Ongenae, and Emanuele Della Valle. 2019. C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328905.3329502

1 INTRODUCTION

Data stream intensive domains, such as the Internet of Things (IoT) and social media, are still gaining popularity. Huge amounts of frequently changing data are continuously produced [3, 7]. However, to extract meaningful insights from multiple heterogeneous data streams, these streams should be combined and integrated with domain knowledge [11].

For instance, industrial IoT is about deploying sensors on production lines to continuously monitor temperature, pressure, vibrations and hundreds of other types of observations about the production tools deployed along the line[1]. On those industrial settings, it is easy to observe throughputs of MB per second (which means GB per hour)[2]. Both the observations and the tools are often classified using taxonomies. For instance, a taxonomy may tell that a pneumatic drill is as a power drill, thus a drill, thus a tool, and thus an instrumentation, etc. All this background knowledge is useful to meaningfully analyze the time-series of observations at-rest, but it challenges real-time analytics. A real-time analysis, willing to aggregate observations about drills, implicitly requires to collect also observations about power drills and pneumatic drills. Naïve implementations may *simply* register multiple queries and union the resulting stream, but this is a resource-aggressive and human-intensive approach. It would be better when the user declares *only the most abstract query* (e.g., observations about drills) and a system takes care of efficiently solving the task (e.g., looking for all the specific types of drills).

Semantic Web technologies are valuable tools to combine various heterogeneous data and integrate it with the domain knowledge [7, 19, 23]. Stream Reasoning (SR) is the research domain that investigates how to infer implicit facts about rapidly changing data through reasoning techniques, such as found in the Semantic Web [10]. RDF Stream Processing (RSP), a sub-domain of SR, focuses on the integration of highly volatile Resource Description Framework (RDF) streams with background knowledge and can continuously answer SPARQL Protocol and RDF Query Language (SPARQL) queries while performing simple reasoning.

The need for SR is rising as data stream production increases and the need for real-time analytics over heterogeneous streams keeps growing. The current state-of-the-art in RSP has mainly focused on query answering over RDF streams [6, 18] while more expressive incremental reasoners [22, 28] have focused on providing expressive reasoning capabilities over slower changing data. However, to provide generic query answering, RSP engines should

DEBS '19, June 24–28, 2019, Darmstadt, Germany

ACM ISBN 978-1-4503-6794-3/19/06…$15.00
https://doi.org/10.1145/3328905.3329502

[1] Interested readers can learn more on https://opcdatahub.com/WhatIsOPC.html

[2] A typical process industry deployment with 200 sensors, which record 20 measurements in 32 bytes messages every 200 ms, generates 0.61 MB/sec (2 GB/hour) per machine. In the oil & gas industry, the number of sensors can easily grow up to hundreds of thousands considering all the machines.

provide some reasoning capabilities [14]. Even hierarchical reasoning capabilities, such as subclass and subproperty reasoning increase the expressivity of the query extensively and simplifies data integration.

Currently, there are three approaches to perform reasoning, each with their own drawbacks:

- Materialization: the process of computing all possible inferences, such that the query can be evaluated without reasoning. Therefore, it also produces data that is not relevant for the Query Answering (QA), resulting in many unnecessary computations and redundant statements. Incremental approaches allow to maintain the materialization in a streaming context, however, this can be very expensive depending on the number of changes in the data [5]. The approach pays off when multiple queries consume the materialized stream.
- Goal driven: relies on backward reasoning to infer only what is relevant for the QA. However, backward chaining causes the same intermediate results to be produced over and over again, resulting in redundant computations [26]. Furthermore, in a streaming context, many recomputations occur since there is no incremental approach possible over the data stream.
- Query Rewriting: is the process of injecting the logic inside the query. This results in a query with multiple UNION clauses. However, UNION is not supported in most of the Stream Processors on which RSP engines can rewrite. To solve this problem, multiple parallel queries are registered [8]. However, empirical evidence shows that the throughput is inversely proportional to the number of queries.

So even for simple reasoning tasks, such as instance checking over hierarchies of classes and properties, each of these techniques has some serious drawbacks. Furthermore, as data stream production keeps rising, current RSP engines are at risk of failing to perform the reasoning at the required throughput[3].

Information Flow Processors (IFPs), such as Complex Event Processing (CEP) engines and Data Stream Management System (DSMS), often support hierarchical reasoning as a standard language feature. Note that the language feature defines hierarchies over relational data and is not related to RDF. However, if the underlying system inherently understands hierarchies, this could be beneficial for each of the mentioned approaches. Namely, it would result in less unnecessary statements, less recomputations and less queries for the materialization, goal driven and query rewriting approaches respectively.

This language feature was never before exploited in stream reasoning, since current approaches either pipelined the IFP with a SPARQL engine [6] or integrated limited amount of stream processing inside the reasoner [22]. Furthermore, even though this feature seems very interesting to exploit, its semantics were never formalized, only defined by implementation.

Therefore, in this paper we formalize continuous hierarchical reasoning and introduce *C-Sprite*, an optimized hierarchical reasoning algorithm that operates in constant time and scales linearly in the number of continuous queries. We present two implementations of C-Sprite: one exploiting the hierarchical features found in existing IFP and one fully optimized based on the theoretical formalization.

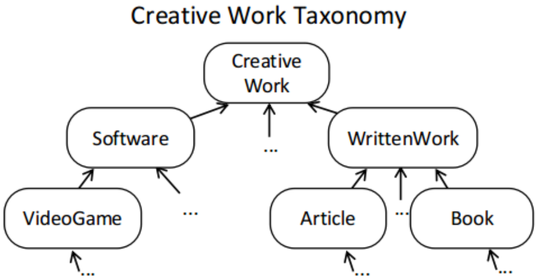


Figure 1: Hierarchical structure of Wikipedia categories

In this paper we tackle the following **Research Question:**

(1) Can we formalize continuous hierarchical reasoning?
(2) Can we exploit the formalization to speed up continuous RSP querying under hierarchical entailment?

We summarize the main **Contributions** as:

(1) Continuous Taxonomy-based Relational Algebra (C-TRA), the continuous extension of the existing Taxonomy-based Relational Algebra (TRA) [20] model. TRA is itself an extension of Relational Algebra and introduces taxonomies to relax query answering in relational databases.
(2) The formalization of the hierarchical reasoning through the means of C-TRA.
(3) An optimized hierarchical reasoning algorithm, i.e. C-Sprite.
(4) An empirical study that validates the approach.

Paper organization: Section 2 introduces an example that will be used throughout the paper. In Section 3 all necessary background is introduced to understand the remainder of the paper. Section 4 and 5 formalize the approach, while in Section 6 we provide a possible data structure and algorithm to efficiently perform the hierarchical reasoning in continuous query answering. Section 7 discusses the related work and in Section 8 we provide the empirical study to show the feasibility of the approach. Section 9 discusses the contributions and Section 10 elaborates on the limitations of the approach and concludes the paper.

2 RUNNING EXAMPLE

Suppose we are interested in retrieving all Wikipedia changes in *creative work*-related articles. Wikipedia exposes the changes that are made as a data stream, detailing the changes to each article and the category it is contained in[4]. These categories are very specific, e.g. videogame, novel, article, etc. However, it is not straightforward to target the categories that should be considered *creative works*.

By introducing a hierarchical description of the various categories, it is possible to define how the categories relate on a hierarchical level. Utilizing a system that understands this hierarchy, one can query the change stream for changes in creative work-related articles and retrieve the specific underlying changed articles, without the need to query all the categories separately. Figure 1 visualizes the hierarchy for the *creative work* categories.

[3] In the industrial IoT example, each machine produces 20k measurements per second. Each measurement is typically described by at least two RDF triples. In the evaluation, we will see that current RSP engines have a maximum throughput of about 60k triples per second.

[4] As defined in https://www.mediawiki.org/wiki/API:Recent_changes_stream.

Throughout the remainder of the paper, we will introduce examples based on this creative work taxonomy.

3 BACKGROUND

This section introduces the background material necessary to understand the remainder of the paper.

3.1 RDFS entailment

In our approach, we focus on hierarchical reasoning, i.e. reasoning over hierarchies of classes and properties. RDF Schema (RDFS) entailment defines 13 rules[5] to express, among others, hierarchical reasoning but also domain/range reasoning and schema reasoning. We focus specifically on the entailment rules rdfs7 and rdfs9 since they specify the hierarchical reasoning we are interested in:

- rdfs7 states that if p is a subproperty of q and a and b are connected through a property p, then the property q holds between a and b:
  rdfs7: $\frac{(p \quad subPropertyOf \quad q) \quad (a \quad p \quad b)}{(a \quad q \quad b)}$
- rdfs9 states that if A is a subclass of B and a is of the type A, then it holds that a is of the type B:
  rdfs9: $\frac{(A \quad subClassOf \quad B) \quad (a \quad type \quad A)}{(a \quad type \quad B)}$

We note that there exist rules with respect to the transitive properties of the subPropertyOf/subClassOf. However, they are not important in instance checking as the transitivity can be obtained by the execution of a sequence of rdfs7/9 rules.

Example 3.1. As shown in Figure 1 *VideoGame* is a subclass of *Software* and *Software* is a subclass of *CreativeWork*. When we have an instance, i.e. *Doom*, of the type *VideoGame* (Doom *type* VideoGame) and execute the rdfs9 rule we obtain that *Doom* is also a *Software*:

$$\frac{(VideoGame \quad subClassOf \quad Software)(Doom \quad type \quad VideoGame)}{(Doom \quad type \quad Software)}$$

Since *Software* is also a subclass of *CreativeWork* and we now know that *Doom* is a *Software*, we obtain through similar means that *Doom* is also a *CreativeWork*:

$$\frac{(Software \quad subClassOf \quad CreativeWork)(Doom \quad type \quad Software)}{(Doom \quad type \quad CreativeWork)}$$

Definition 3.2. The **materialization** of a knowledge base under RDFS entailment is the process of computing and storing all the inferred facts derived from the executing of the RDFS rules. The materialization stops when no new facts can be derived from the execution of the RDFS rules.

Example 3.3. (cont'd) The materialization of the knowledge base containing only the triple ($Doom\ type\ VideoGame$) and the rdfs9 rule, according to the schema depicted in Figure 1, results in the triples: ($Doom\ type\ VideoGame$), ($Doom\ type\ Software$), ($Doom\ type\ CreativeWork$).

3.2 SPARQL under RDFS9 entailment

SPARQL is the query language for RDF data[6], different from other QA systems, it can match data that is not explicitly stated, but can be derived under a certain entailment[7]. More specifically, implicit data can be derived from the given data, the ontology and an ontological language (or entailment).

Definition 3.4. We define the evaluation of SPARQL under RDFS9 entailment as $eval(G, BGP, O, RDFS9)$ with $RDFS9$ the entailment regime, O the ontology, BGP the basic graph pattern used in the SPARQL query and G the RDF dataset.

Example 3.5. Lets consider again our simple dataset containing the single triple $G = \{(Doom\ type\ VideoGame)\}$. We are interested in querying for all *CreativeWork* concepts, as defined in the ontology hierarchy in Figure 1. The BGP consists thus of "?w type CreativeWork". The ontology O is the ontology represented by the hierarchical definition of concepts as depicted in Figure 1 and the entailment is the RDFS entailment consisting of the rdfs9 rule. When evaluating the query without the ontology and the entailment regime only the explicit data can be queried and no matches are found: $eval(G, BGP, \emptyset, \emptyset) = \emptyset$
When considering the ontology and the entailment regime, we can find a match through the derivation of the implicit data as described in Example 3.1 (we derive that *Doom* is a $CreativeWork$) while executing the query: $eval(G, BGP, O, RDFS9) = \{?w : Doom\}$
Another option is to first materialize the dataset and then evaluate the query without the need for the entailment regime during the query evaluation. First, we obtain the materialize dataset: (G_{rdfs9}=$\{(Doom\ type\ VideoGame), (Doom\ type\ Software), (Doom\ type\ CreativeWork)\}$). Then we can evaluate the query without the entailment regime: $eval(G_{rdfs9}, BGP, \emptyset, \emptyset) = \{?w : Doom\}$

3.3 Stream Processing

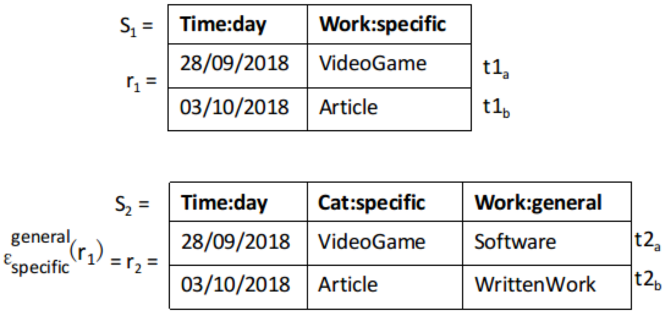
We introduce a window operator to be able to process the content in the stream, based on the definitions from CQL [2] and RSP-QL [13].

Definition 3.6. A **window** $W(S)$ is a set of data extracted from a stream S. A **time-based window** is defined based on two time instances o and c, respectively the opening and closing time instant, such that: $W(S) = \{d | (d, t) \in S \wedge t \in (o, c]\}$. With d all data in S at a specific time instant.

The window operator allows us to extract defined and processable chunks of data from the unbounded data stream. A time-based window defines the content of a window based on a certain amount of time that passes. A time-based sliding window extends this notion in such a way that the window slides through time, to have overlapping windows.

Definition 3.7. A **time-based sliding window operator** $\mathbb{W}$ is defined based on three parameters (α, β, t^0), such that α is the width of the window, β is the slide and t^0 is the time instant on which $\mathbb{W}$ starts to operate. The sliding window operator produces a sequence of time-based windows $W_1, W_2, \ldots$ such that: 1) the opening of the first window (W_1) is t^0; 2) each window has width α, i.e. window W_i is defined through (o_i, c_i) with $c_i - o_i = \alpha$; 3) β is the differences between the opening times of two consecutive windows, i.e., the difference between the opening time o_{i+1} of W_{i+1} and o_i of W_i is β.

[5] https://www.w3.org/TR/rdf11-mt/#rdfs-entailment
[6] https://www.w3.org/TR/rdf-sparql-query/
[7] https://www.w3.org/TR/sparql11-entailment/


S₁ =
r₁ =
t1ₐ
t1_b
S₂ =
ε general specific (r₁) = r₂ =
t2ₐ
t2_b

Time:day	Work:specific
28/09/2018	VideoGame
03/10/2018	Article

Time:day	Cat:specific	Work:general
28/09/2018	VideoGame	Software
03/10/2018	Article	WrittenWork

Figure 2: T-relation, t-schema and upward extension TRA example.

3.4 Hierarchical Stream Processing Languages

IFP often include a hierarchical feature in their language [15]. For instance, the Event Processing Language (EPL) used in Esper[8] allows to define the hierarchies of events in its language. Listing 1 shows an example of defining a small part of the category taxonomy from Figure 1 in EPL.

Listing 1: Hierarchical definition in EPL

```
create schema CreativeWork(id string, ts double);
create schema Software() inherits CreativeWork;
```

For other examples, we direct the reader to Eckert et al. [15].

3.5 TRA

The Taxonomy-based Relational Algebra (TRA) [20] introduces taxonomies to relax query answering in relational databases. We introduce some of the key concepts of TRA, since they will be used later in the formalization of C-Sprite.

First, we define an h-domain which defines hierarchies and taxonomies.

Definition 3.8. An **h-domain h** is composed of:

- a finite set $L = \{l_1, ..., l_k\}$ of levels, each associated with a set of values, i.e. the *members* of the level and denoted by $M(l)$;
- a partial order $\leq_L$ on L having a bottom ($\perp_L$) and a top element ($\top_L$).
- a family of functions $LMAP_{l_1}^{l_2} : M(l_1) \rightarrow M(l_2)$, called level mappings.

A **taxonomy** is a set of h-domains.

Example 3.9. In our running example from Section 2 we can create a set of levels $L = \{creativework, general, specific, ...\}$ with $M(creativework) = \{CreativeWork\}$,
$M(general) = \{Software, WrittenWork, ...\}$,
$M(specific) = \{VideoGame, Article, Book, ...\}$.

Besides ordering between levels, there is also an ordering between members:

Definition 3.10. Let h be an h-domain and m_1 and m_2 are members of respectively l_1 and l_2. There exists an **ordering on the members** $m_1 \leq_M m_2$ if $l_1 \leq_L l_2$ and $LMAP_{l_1}^{l_2}(m_1) = m_2$.

We can now define a schema over taxonomies and t-relations, as the natural extension of a relation table built over taxonomy defined values:

Definition 3.11. Let T be a taxonomy. A **t-schema** (schema over taxonomies) for T, is denoted by $S = \{A_1 : l_1, ..., A_k : l_k\}$, with A_i the attribute name of the *h-domain* and l_i the level of some *h-domain* in T.

Example 3.12. Figure 2 depicts the t-schema $S_1 = \{Time : day, Cat : specific\}$.

We can now define a tuple and relation over a taxonomy:

Definition 3.13. A **t-tuple** over a *t-schema* $S = \{A_1 : l_1, ..., A_k : l_k\}$ for a taxonomy T is a function mapping each attribute A_i to a member of l_i. A **t-relation** r over S is a set of *t-tuples* over S.

Example 3.14. In Figure 2 we can see the t-tuples $t1_a$ & $t1_b$ in the t-relation r_1.

Last but not least, we introduce the upward extension operator that allows to take the taxonomy into account:

Definition 3.15 (upward extension). Let r be a *t-relation* over S, A an attribute in S defined over a level l, and l' a level such that $l \leq_L l'$. The *upward extension* of r to l', denoted by $\varepsilon_{A:l}^{A:l'}(r)$, is the *t-relation* over $S \cup \{A : l'\}$ defined as:
$\varepsilon_{A:l}^{A:l'}(r) = \{t | \exists t \in r : t[S] = t', t[A : l] = LMAP_l^{l'}(t'[A : l])$

Example 3.16. Figure 2 depicts the upwards extensions $\varepsilon_{specific}^{general}(r_1)$ in r_2.

Besides the upward extension, TRA also provide downward extension, upward/downward selections, projections, unions, differences and joins, which are omitted because they are not relevant for the remainder of the paper. We note that the TRA upward extension and the rdfs9 rule are alternative formalisms to capture the same idea.

Definition 3.17. **TRA-** is the subset of TRA without the downward extension, join and difference operators.

In the remainder of the paper, we will assume the usage of TRA-.

4 FROM TRA TO SPARQL UNDER ENTAILMENT

In our approach, we want to formalize the semantics of the hierarchical reasoning inside the IFP, which is built upon Relational Algebra (RA). This can be achieved by extending the RA inside the IFP to include hierarchies, which is exactly what TRA does. Therefore, in this section, we align our approach with TRA.

4.1 TRA for Ontologies

We first describe how we can align TRA with ontologies, we limit the ontological language to the definition of classes and properties w.r.t. RDFS rules 7 and 9, thus the hierarchical definitions of classes and properties.

[8] http://www.espertech.com/esper/

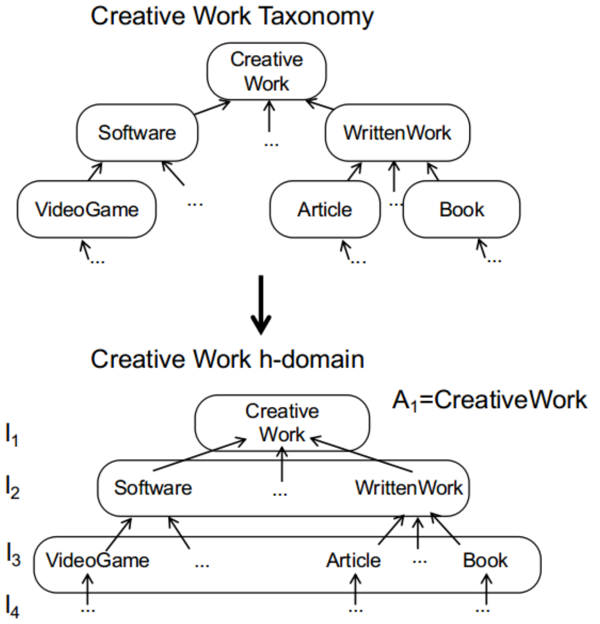
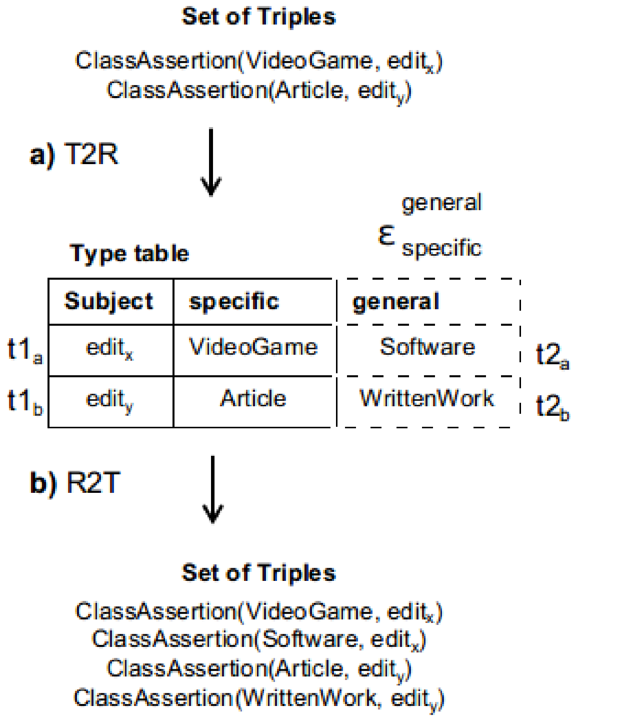


Figure 3: Example ontology and taxonomy alignment



ClassAssertion(VideoGame, $edit_x$)
ClassAssertion(Article, $edit_y$)

	Subject	specific	general	
$t1_a$	$edit_x$	VideoGame	Software	$t2_a$
$t1_b$	$edit_y$	Article	WrittenWork	$t2_b$

ClassAssertion(VideoGame, $edit_x$)
ClassAssertion(Software, $edit_x$)
ClassAssertion(Article, $edit_y$)
ClassAssertion(WrittenWork, $edit_y$)

Figure 4: Alignment of a) triples with relation data and b) back to triples.

4.1.1 Alignment of taxonomies with ontologies. Since ontologies and TRA have a different data model, we first describe how they can be aligned. TRA starts from the assumption of *levels* and *members* that is missing in ontologies. However, we can introduce the notion of *levels* by visualizing the ontology classes as one or more trees and assigning all classes that have the same path length from the root to the same *level* in an h-domain. The *members* of the *levels* are the ontology classes themselves. The ordering between the classes is maintained by the ordering between the levels and the members. This is depicted in Figure 3. Note that when multiple inheritance occurs in the ontology, multiple h-domains are created. For example, RSP is a subclass of both the *Semantic Web* and *Stream Processing*. This results in two h-domains, one with RSP in a sublevel of *Semantic Web*, and one with RSP in the sublevel of *Stream Processing*. The alignment for the ontology properties is similar.

4.1.2 Alignment of triples with Relational Data. Since we focus on hierarchical reasoning, we limit our discussion to two types of triples, i.e. class assertions and object property assertions. We utilize the Manchester syntax[9] for this purpose: class assertions (i.e. ClassAssertion(C,s), with *C* an ontology class and *s* an individual) and object property assertions (ObjectPropertyAssertion(s,P,o), with *P* an object property and *s* and *o* individuals). For simplicity we focus on the class assertions, however, the definitions for object property assertions are straightforward.

Definition 4.1. We define the function **T2R**: $\{triples\} \rightarrow R$ that maps a set of triples to relational data through the use of mappings. Each class assertion triple (i.e. $ClassAssertion(C_i, x)$) has a relational presentation where the schema consists of $S = \{Subject, A_j : l_k\}$ with *Subject* the individual name, A_j the taxonomy attribute of C_i and l_k the level of C_i in the taxonomy. Adding a new class assertion (e.g. $ClassAssertion(C_q, x)$) results in updating the schema by adding a new column to store the additional type.

Example 4.2. The class assertion ($ClassAssertion(Article, edit_x)$) translates to the first row ($t1_a$) of Figure 4 a).

4.1.3 Alignment of Relational Data with triples. We can now going in the other direction and align the relational data with triples.

Definition 4.3. We define a function **R2T**: $R \rightarrow \{triples\}$ that maps relational data (obtained by *T2R*) to triples through the use of mappings. Each tuple in $S = \{Subject, A_j : l_k\}$ results in a triple ($ClassAssertion(C_i, x)$) with $t[A_j : l_k] = C_i$. When the schema contains multiple columns (e,g, $S_2 = \{Subject, A_j : l_k, A_p : l_q\}$) then multiple triples are generated.

Example 4.4. The first row (after extending the table through the upward extension $\varepsilon^{general}_{specific}$) ($t2_a$) of Figure 4 b) translates to the class assertions:
$ClassAssertion(Article, edit_x), ClassAssertion(WrittenWork, edit_x)$.

The functions *R2T* and *T2R* are also known as *direct mappings* and its inverse application. We refer the interested reader to Sequeda et al. [27] for more a more detailed description.

[9] https://www.w3.org/TR/owl2-manchester-syntax/

4.2 Alignment of SPARQL under RDFS entailment with TRA

Now that we have aligned ontologies with TRA, we can further formalize our approach by aligning with SPARQL under entailment.

Theorem 4.5. *The SPARQL evaluation of a dataset under RDFS9 entailment and a dataset under upward extension are equal.*

Proof. As the evaluation of SPARQL under entailment can be implemented as first materializing the dataset and then evaluating the query, we need to prove:

$$eval(G_{rdfs9}, BGP, \varnothing, \varnothing) = eval(G_{\varepsilon}, BGP, \varnothing, \varnothing) \tag{1}$$

Through the materialization, we can further limit the proof to the alignment of the dataset under RDFS9 entailment and upward extension: $G_{rdfs9} = G_{\varepsilon}$.

Figure 5 shows how the use of TRA's upwards extension compares to RDFS9.

Assumptions:

(1) We assume that the ontology O contains a hierarchy of a certain number of subclasses: $\exists C_0, .., C_{k+1} \in O : C_i \sqsubseteq C_{i+1} \wedge level(C_i) \leqslant level(C_{i+1})$ with $i <= k$ and $level(C_j)$ the mapping of each ontology class to a certain level in the h-domain.

(2) There is a function $T2R$ ($R2T$) that maps triples to relation data (relation data to triples), as described in Section 4.1.2.

(3) There is a function $rdfs9_{i..j}$ that applies the sequence of RDFS9 rules[10] $(C_i \quad subclassOf \quad C_{i+1}), .., (C_{j-1} \quad subclassOf \quad C_j)$

(4) n is the depth of the hierarchy used in the reasoning.

Since RDFS9 entailment over a dataset equals the union of the entailment on each triple in the dataset [29], we can simplify the proof for a single triple.

We prove that $rdfs9(x \in C_i) = R2T(\varepsilon(T2R(x \in C_i)))$ for a certain ontology O, with $x \in C_i = ClassAssertion(C_i, x)$:

Base case (n=1): In this case we apply one hierarchical reasoning step. This means that $C_0 \sqsubseteq C_1 \wedge level(C_0) \leqslant level(C_1)$ while its known that $x \in C_0$. For readability we show the base case for $C_0 \sqsubseteq C_1$ but it holds for every i such that $C_i \sqsubseteq C_{i+1}$.

We need to prove that: $rdfs9_{0..1}(x \in C_0) = R2T(\varepsilon^{l_1}_{l_0}(T2R(x \in C_0)))$

Applying the RDF9 entailment we obtain that $x \in C_1$:

$$\frac{(x \quad type \quad C_0)(C_0 \quad subclassOf \quad C_1)}{(x \quad type \quad C_1)} rdfs9 \tag{2}$$

Through the upward extension we maintain a table with an additional column:

$$\varepsilon^{l_1}_{l_0}(r) = r \cap T2R((x \quad type \quad C_1)) \tag{3}$$

By the definition of $R2T$ and $T2R$ we can conclude that the result of (2) equals R2T(3).

Inductive hypothesis (n=k): We assume that the theorem holds for all values of n up to some $k, k \geq 0$. With k the difference in hierarchy level.

$rdfs9_{0..k}(x \in C_0) = R2T(\varepsilon^{l_k}_{l_0}(T2R(x \in C_0)))$

Inductive step (n=k+1): Lets assume that the hierarchy is of size $k + 1$.

$$\begin{aligned} & rdfs9_{0..k+1}(x \in C_0) \\ & \quad = R2T(\varepsilon^{l_{k+1}}_{l_0}(T2R(x \in C_0))) \\ & \quad = R2T(\varepsilon^{l_{k+1}}_{l_k}(\varepsilon^{l_k}_{l_0}(T2R(x \in C_0)))) \text{ def } \varepsilon \\ & \quad = R2T(\varepsilon^{l_{k+1}}_{k}(T2R(rdfs9_{0..k}(x \in C_0))) \text{ inductive step} \\ & \quad = rdfs9_{k..k+1}(rdfs9_{0..k}(x \in C_0)) \text{ base case} \\ & \quad = rdfs9_{0..k+1}(x \in C_0) \text{ def transitivity rdfs9} \end{aligned}$$

Q.E.D.

□

The proof for rdfs7 was omitted, as it is similar to the proof for rdfs9. In Figure 5 we have windowed the data and used the RStream function to assign timestamps to the resulting solution mappings. The RStream function allows you to stream out the obtained answers. We refer the interested reader to Arasu et al. [2] for more information. The incorporation of the streaming operators is further detailed in Section 5.

5 C-TRA: CONTINUOUS TRA

Now that we have aligned our approach in a rather static context, we formalize the applicability in a streaming environment. Therefore, we extend TRA, which has been build for static environments, to Continuous TRA (C-TRA).

To extend TRA to C-TRA, we rely on the "black box" components of CQL [2] that state that instead of integrating the streaming operators in the algebra, it is possible to convert the stream to a relational form through a Stream-to-Relation (S2R) operator. The remaining operations can be performed in relational form, allowing to exploit well-understood relational semantics. CQL can be composed of a S2R operator, followed by well-known relation algebra operations and a Relation-to-Stream (R2S) operator to stream out the results:

$$CQL = S2R + RA + R2S \tag{4}$$

Defining a Stream-to-Stream (S2S) operator in CQL is done by combining a S2R operator with a Relation-to-Relation (R2R) operator that exploits relation algebra and an R2S operator:

$$S2S_{CQL} = S2R + R2R_{RA} + R2S \tag{5}$$

5.1 A continuous taxonomy query language

A continuous taxonomic query language (CTQL) that takes taxonomies into account can be defined as standard CQL, but operating on TRA instead of standard RA:

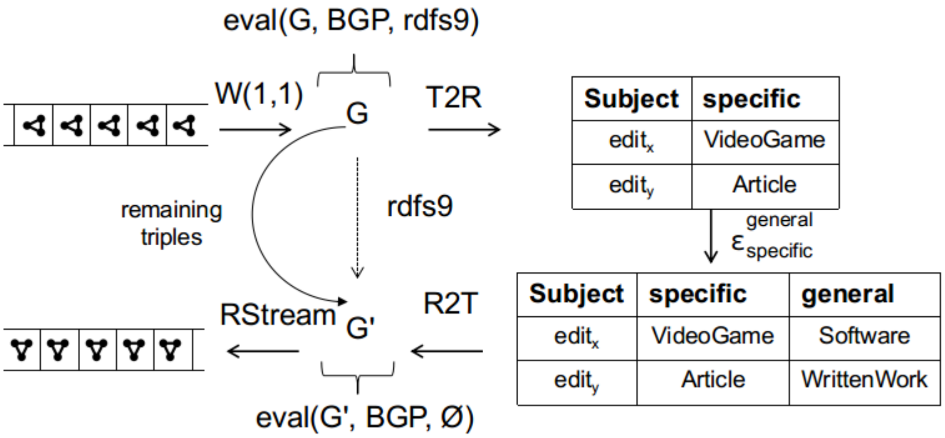
$$CTQL = S2R + TRA + R2S \tag{6}$$

The S2S operator over CTQL can then be defined as[11]:

$$\begin{aligned} S2S_{CTQL} &= S2R + R2R_{TRA} + R2S & (7) \\ &= \mathbb{W}(\alpha, \beta, t^0) + R2R_{TRA} + RStream & (8) \end{aligned}$$

[10] The semantics of the transitive property of subclassof is the same as the sequental excecution of RDFS9.

[11] Note that there are many options to perform the S2R operation. We opted for a sliding window and leave the further generalization for future work.



Subject	specific
edit$_x$	VideoGame
edit$_y$	Article

Subject	specific	general
edit$_x$	VideoGame	Software
edit$_y$	Article	WrittenWork

Figure 5: Flow of using TRA's upward extension compared to RDFS9.

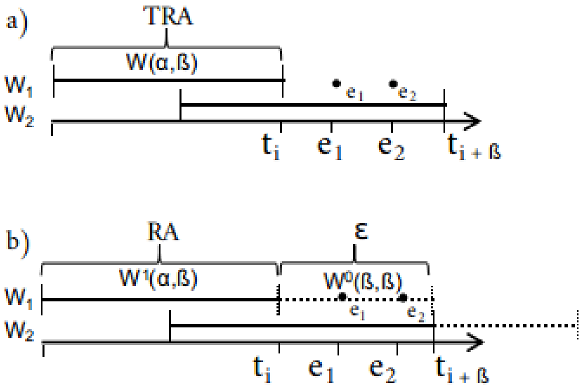


Figure 6: Window sequence timeline of a) TRA and b) $\varepsilon(RA)$. In a) events e_1 and e_2 need to wait for the shifting of the window (W_2) to be taken into account, while in b) this period is used to perform the upward extension.

We can further decompose this formula since $TRA = \varepsilon(RA)$. Figure 6 a) visualizes a timeline that illustrates the usage of TRA and Figure 6 b) its further decomposition. The figure shows that the events e_1 and e_2, arriving between the shifting of the window, can be upward extended in an additional window, exploiting the wait time in between windows. The decoupling of the upwards extension form the RA allows to perform the upward extension in between window shifts while the RA is executed upon the window. We define this decoupling more formally:

$$S2S_{CTQL} = \mathbb{W}(\alpha, \beta, t^0) + R2R_{TRA} + RStream \tag{9}$$

$$= \mathbb{W}(\alpha, \beta, t^0) + R2R_{\varepsilon}(R2R_{RA}) + RStream \tag{10}$$

$$\begin{aligned} &= \mathbb{W}^0(\beta, \beta, t^0) + R2R_{\varepsilon} + RStream \\ &\quad + \mathbb{W}^1(\alpha, \beta, t^0) + R2R_{RA} + RStream \end{aligned} \tag{11}$$

with $\mathbb{W}_0(\beta, \beta, t^0)$ opening at the previous evaluation of $\mathbb{W}^1(\alpha, \beta, t^0)$ and closes on the next evaluation. This enforces that each window's t^0, i.e. the time instant on which each window $\mathbb{W}^i$ starts to operate, are synchronized.[12]

This shows that we can extract the extension and execute it before the rest of the processing, while we can rely on RA for the further processing steps. Furthermore, it allows to eliminate data early on, when they do not meet the hierarchical conditions.

5.2 C-TRA for Stream Reasoning

Since IFP operate on RA, we extended the use of RA to TRA. We have shown that TRA aligns with ontologies and that C-TRA can be used in a streaming fashion. Furthermore, the decomposition described in C-TRA allows to perform the hierarchical reasoning before the RA. This means that we can perform our triple based hierarchical reasoning inside a IFP and perform the reasoning before other operations such as joins or aggregations. This allows to eliminate triples early on based on the hierarchical requirements in the query.

6 EFFICIENT HIERARCHICAL REASONING

This section discusses how we can efficiently store and retrieve the hierarchy for QA. We begin by describing a data structure for storing the data and the queries and then we discuss the algorithm for querying and study its complexity.

6.1 Data structure

A possible data structure for efficient lookup of the parent classes for a specific class in the hierarchy is by saturating the hierarchy and storing for each class a list of all the parents, as visualized in Figure 7 a). By storing the list of parents in a hashmap, using the class name as the key and storing the list of parents as the value, one can look up the parents for a specific class in constant time ($O(1)$).

Since we need a way to efficiently query the data, we create a new instance of the hierarchy that only contains the concepts

[12] In the remainder of the paper we will focus on a window $\mathbb{W}^0(1, 1, t^0)$ without losing generality, but relaxing the need for synchronizing t^0.

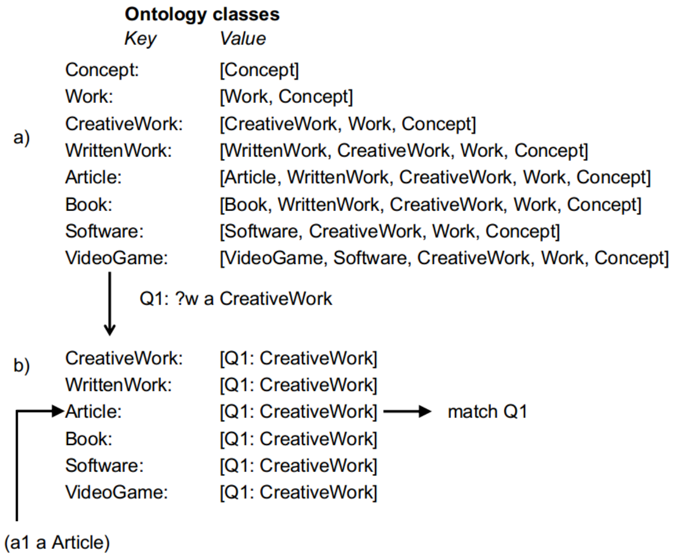

Ontology classes
Key
Value
a)
Concept: [Concept]
Work: [Work, Concept]
CreativeWork: [CreativeWork, Work, Concept]
WrittenWork: [WrittenWork, CreativeWork, Work, Concept]
Article: [Article, WrittenWork, CreativeWork, Work, Concept]
Book: [Book, WrittenWork, CreativeWork, Work, Concept]
Software: [Software, CreativeWork, Work, Concept]
VideoGame: [VideoGame, Software, CreativeWork, Work, Concept]
Q1: ?w a CreativeWork
b)
CreativeWork: [Q1: CreativeWork]
WrittenWork: [Q1: CreativeWork]
Article: [Q1: CreativeWork]
match Q1
Book: [Q1: CreativeWork]
Software: [Q1: CreativeWork]
VideoGame: [Q1: CreativeWork]
(a1 a Article)

Figure 7: Flow of the algorithm

(keys) that have the queried type in their list of parent concepts. When the concepts have been filtered, we link each concept to the query. When multiple queries are added, each concept contains a list of queries it matches according to the hierarchy. We focus specifically on queries asking for specific type instances (queried types). In Figure 7, query Q1 asks for all instances of *CreativeWork* related categories. The concepts that are not *CreativeWorks*, such as *Concept* and *Work*, are dropped and a direct link is made to the query.

When new data arrives, such as an *Article* in Figure 7, a simple lookup in the hashmap allows us to detect that query Q1 matches.

6.2 Algorithm

Let us define an algorithm to query hierarchical classes encoded on the data structures introduced above. Algorithm 1 shows the pseudo-code that describes how the data structure is constructed to efficiently perform the querying. First, we convert the ontology hierarchy in a hashmap H containing for each class all its parents. Each time a query is registered, a copy of the hierarchy is pruned such that it only contains the concepts that have the queried type in their list of parents. The selected concepts are then directly linked to the queries. This allows to perform the hierarchical reasoning as a lookup in a hashmap.

Algorithm 2 is executed on the ingestion of a new *triple*. When a new triple is received, the system executes the *CheckHierarchyMatch* function that takes the triple and the pruned hierarchy hashmap as arguments. By looking up the asserted types of the triple in the hashmap, it detects which queries the triples match.

6.2.1 Complexity study: Let's assume that the number of queried classes in all the queries is m (i.e. $m = \sum_{i=0}^{len(Q)} len(Q_i)$). Thus, the complexity of first looking up in the pruned hashmap if the triple's type matches any queries and then iterating over them is $O(1) + O(m)$). The complexity only depends on the number of queries, which is typically low. Indeed, for each triple in the stream the execution is performed in constant time.

Algorithm 1 Query registering

```
Precondition: $Q$ a collection of queries, each interested in one or more types.
$H \leftarrow ConvertToHierarchy(O)$   ▷ Stores parents for each class in the Ontology $O$
function PrepareHierarchy($H, Q$)
    $H' \leftarrow []$
    for $q \in Q$ do
        for $(concept, parents) \in H$ do
            if $q \in parents$ then
                $H'[concept].append(q)$
            end if
        end for
    end for
    return $H'$
end function
```

Algorithm 2 Calculate the query matches on a hierarchical level

```
Precondition: $Q$ a collection of queries, each interested in one or more types.
$H \leftarrow ConvertToHierarchy(O)$   ▷ Stores parents for each class in the Ontology $O$ (preprocessing step)
$H' \leftarrow PrepareHierarchy(H, Q)$   ▷ (preprocessing step)
$triple \leftarrow$ ClassAssertion(type,subject)
function CheckHierarchyMatch($H', triple$)
    $QueryMatches \leftarrow H'(types(triple))$   ▷ types extracts the type assertions of a triple
    return $QueryMatches$
end function
```

6.3 Definition in EPL

If we want to exploit the hierarchical reasoning inside the IFPs, we need to align the triples with the IFPs events. One possible way to achieve this is by defining the class and property names as event definitions. For example, the class assertion triple *(ClassAssertion(Article,edit))* becomes *Article(edit)*, (with *Article* an event definition and *edit* a parameter) and similar for the property assertions. This way *Article* is an event definition that can exploit the hierarchy.

7 RELATED WORK

In this section, we elaborate on the related approaches in the literature that are able to perform hierarchical reasoning (or more) and describe how they compare to C-Sprite. Table 1 summarizes the related work.

Compared to C-Sprite, current approaches suffer from the problems that accompany materialization, backward chaining or query rewriting. Either they infer too many triples, perform redundant computations or suffer from a large number of queries.

The C-SPARQL engine [6] builds on existing IFP and SPARQL engines to respectively perform the windowing of the streams and the querying of the data captured in the window. C-SPARQL supports reasoning and querying through the use of Jena[13]. However, C-SPARQL is pluggable, allowing the support of other reasoners.

EP-SPARQL (JTalis)[1] is an event processing enabled SPARQL engine that builds on top of logic programming. To perform reasoning EP-SPARQL supports event-driven backward chaining by relying on Prolog.

SPARKWAVE[17] exploits the Rete algorithm [16] to materialize the RDF streams through an adaption of Rete that also incorporates the query answering. Even though it exploits Rete in a smart way, it is prone to the usual materialization problems, i.e. the inference of many unnecessary triples.

StreamQR [8] enables the execution of continuous queries under entailment by rewriting the queries in multiple parallel queries. CQELS [18] is utilized to execute the rewritten queries. It is a very promising technique, however, the query rewriting easily results in a high number of rewritten queries, drastically lowering the engine's performance.

LiteMat [9] uses an encoding scheme to encode the hierarchies to improve materialization and query rewriting in time and space complexity. The encodings allow to translate the entailment problem to a rewriting problem in terms of filtering the hierarchical entailment as numbers. However, due to the used encoding scheme, LiteMat is not able to encode multiple inheritance. C-Sprite does not encode the hierarchies in a numerical representation, but exploits the hierarchical support of the underlying IFP.

Strider [25] is a distributed RSP engine that exploits distributions techniques to efficiently execute and dynamically update query plans. It focuses specifically on query answering. In an extension, i.e. Strider-R [25], it exploits LiteMat to perform the reasoning.

RDFox [22] is the fastest incremental reasoner currently available, utilizing an optimized version of the Delete and Rederive (DReD) algorithm [30] for efficient incremental reasoning. However, it does not provide any mechanisms to deal with high-volatile data streams.

IMARS[5, 12] keeps an incremental maintenance of the materialized knowledge that is valid within a given window of time. It adapts DReD for its applicability in SR. Even though incremental maintenance of the materialization is more efficient than rematerializing each window, its efficiency is dependent on the percentage of changes in the stream.

G-ToPSS [24] is a subgraph matching algorithm that also exploits hashmaps to efficiently match subgraphs. Taxonomy matching is supported, however, in contrast to C-Sprite, it requires to iterate over all the parents a concept has.

8 EVALUATION

To evaluate the feasibility of C-Sprite, we compared C-Sprite's maximum throughput with other engines when increasing the window size and the size of the ontology used to perform the reasoning. Before jumping to these evaluations, we first describe the used dataset and we explain how we selected the engines we compare against.

The evaluation itself was conducted on a 16 core Intel Xeon E5520 @ 2.27GHz CPU with 12GB of RAM running on Ubuntu 16.04 and utilizing Esper 6.1.0.

8.1 Dataset

DBpedia [4] is Wikipedia content represented as a Semantic Web. DBpedia live [21] provides the Wikipedia changes as structured data, conform to the DBpedia ontology [14]. We have used these changes to re-stream the wikipedia changes as structured data, such that we can control the stream rate in orde to evaluate the throughput of C-Sprite. The evaluation thus consists of querying all the changes to creative works that are happening to DBpedia. The RSP-QL query used within the evaluation is shown in Listing 2.

We loaded all the additions made between November 2013 and May 2018. As we are only interested in querying the types of the concepts, we filtered the triples in the data that did not describe a type assertion. Furthermore, when multiple types have been provided in the data, we only keep the most specific type assertions, such that the hierarchical reasoning is necessary to discover the different *Creative Works* in the data. The final dataset contains more than 3 million triples, of which more than 56 thousand describing *Creative Works*. Table 2 summarizes the characteristics of the used dataset.

8.2 Engines Selected as Terms of Comparison

We have selected C-SPARQL, StreamQR and SPARKWAVE as they are the most prominent RDF stream processors currently available. We also added StreamFox, i.e. a C-SPARQL approach utilizing RDFox instead of Jena. We note that we did not take LiteMat or Strider-R into account as these approaches are not complete due to their encoding scheme and at the time of writing the publicly available code of these tools is incomplete[15]. As some of these approaches can perform more advance reasoning than the hierarchical reasoning discussed in this paper, we carefully adapted their configuration such that the reasoning tasks only consists of hierarchical reasoning such that it is a fair comparison.

Esper implements a similar algorithm as the one we discussed in Section 6. Therefore, we build upon Esper to perform the evaluation. Note that we convert the ontology to EPL rules that can be interpreted by Esper and convert the triples in the stream to Esper events. The approach exploiting Esper's internals is further denoted as *C-SpriteEsper*. As Esper provides more functionality than needed to efficiently perform hierarchical reasoning over data streams and the algorithm is not as optimized as the one described in Section 6, i.e. it does not prune the list of parents based on the registered queries. Therefore, in this evaluation we have added our own algorithm as an upper bound. The latter is denoted as *C-SpriteOpt*[16] and provides the implementation of the optimized algorithm described in Section 6. We can thus consider *C-SpriteEsper* as a lower bound and *C-SpriteOpt* as an upper bound of the possible C-Sprite performance.

[13] https://jena.apache.org/

[14] https://wiki.dbpedia.org/services-resources/ontology

[15] We tried to contact the authors but the main contributor left the research facility.

[16] The source code of CSprite together with the experiment data can be found on https://github.com/pbonte/C-Sprite

Table 1: Comparison C-Sprite and related approaches

	Reasoning	Components	Problems
C-SPARQL	Materialization	IFP + SPARQL engine	Unnecessary triples
EP-SPARQL	Backward Chaining	Prolog	Redundant computations
IMARS	Incremental Materialization	IFP + SPARQL engine	Dependent on changes
SPARKWAVE	Materialization	Rule engine	Unnecessary triples
StreamQR	Query rewriting	Rewriter + CQELS	Many Queries
LiteMat	Encodings	encoding + SPARQL engine	Vector representation
C-Sprite	Hierarchies	inside IFP	Simple entailment

Table 2: Dataset statistics

	Absolute Number	Relative Number
all triples	3.511.629	100%
Creative Works	56.581	1,61%
Top 5 Creative Works:		
MusicalWork	21.438	0,61%
Film	13.890	0,40%
WrittenWork	6.814	0,19%
TelevisionShow	4.579	0,13%
Software	4.493	0,13%

Listing 2: High level query in RSP-QL utilized in the evaluation

```
REGISTER QUERY <http://streamreasoning.org/csprite/s1> AS
PREFIX : <http://streamreasoning.org/csprite/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
SELECT *
FROM NAMED WINDOW :win1 [RANGE 1s, SLIDE 1s]
ON STREAM :dbpediaChanges
WHERE {
 WINDOW ?w {
   ?change rdf:type dbpedia:CreativeWork.
 }
}
```

8.3 Throughput Evaluation

First, we evaluate the maximum throughput by streaming the DBpedia changes at the highest possible rate. This is done by setting up a websocket such that each engine can consume the stream at the highest possible rate. Figure 8 shows the throughput for each of the engines with increasing window sizes. As there are no joins, the window has no significant influence on most engines. C-SPARQL and StreamFox show limited influence as their query engine processes the whole content of the window when the window shifts. The other approaches process each triples as it is injected. We can see that both C-Sprite implementations (i.e., *C-SpriteOpt* and *C-SpriteEsper*) clearly outperform all the other approaches.

Figure 9 depicts the memory consumption during the same experiment. It is clear that the materialization approaches, i.e. C-SPARQL and StreamFox, have an increasing memory footprint when the window increases. This can be expected as more and more triples need to be maintained inside the window. The other approaches are less prone to the memory increase. We see that the memory footprint of SPARKWAVE is even lower than the one of *C-SpriteEsper* but similar to *C-SpriteOpt*. This is due to encoding techniques specially incorporated in SPARKWAVE to lower the memory footprint. Said techniques have not yet been incorporated in C-Sprite. However, the most important message is that C-Sprite's memory consumption is not increasing as with the materialization approaches.

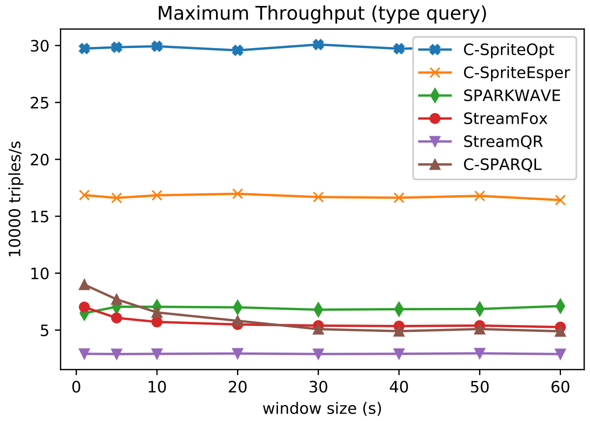


Figure 8: Evaluation of the maximum throughput (more is better) of C-Sprite when increasing the window size, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

8.4 Ontology depth Evaluation

The complexity study in Section 6 clearly shows that the number of parents which a class has, should not influence the complexity of the approach. Therefore we made the DBpedia ontology artificially deeper. This is done by adding artificial subclasses between the *CreativeWork* class and the specific classes used within the stream. With the ontology depth, we mean the length of the path from the root to the classes without children if we visualize the ontology as a tree. Figure 10 shows the influence of increasing ontology depth on the throughput. We can clearly see that the materialization approaches become less performant when the ontology depth increases. This is due to the fact that more triples need to be inferred.

Figure 11 shows the consumed memory while increasing the ontology depth. Almost all the approaches are influenced by the increase in depth. C-Sprite stays rather constant as it is not materializing all the triples. StreamQR and SPARKWAVE show a small increase in memory consumption.

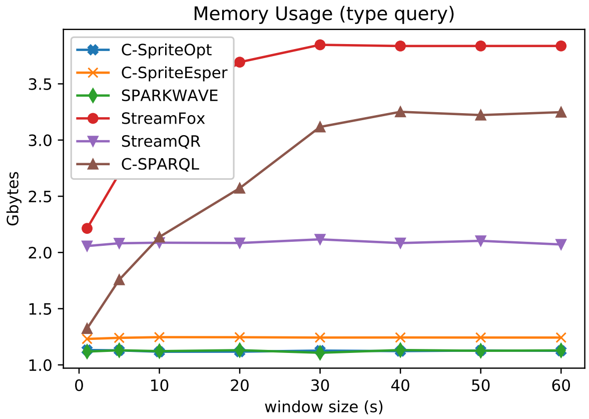


Figure 9: Evaluation of the memory consumption (less is better) of C-Sprite when increasing the window size, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

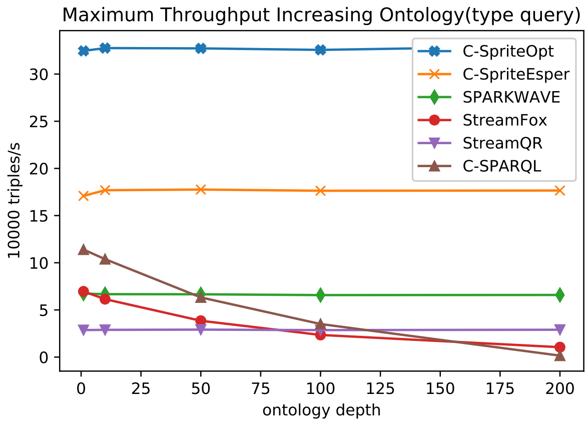


Figure 10: Evaluation of the maximum throughput (more is better) of C-Sprite when increasing the ontology depth, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

We also evaluated the correctness when increasing the ontology depth of the various approaches. Also in this scenario, all the engines produced the correct results.

8.5 Conclusion

It is clear that both C-Sprite implementations outperform the other approaches in terms of maximum throughput. Only SPARKWAVE has a smaller memory footprint than *C-SpriteEsper* due to its special encoding scheme, but similar to it is *C-SpriteOpt*. *C-SpriteOpt* sets a realistic upper bound for the performance while *C-SpriteEsper* sets a possible lower bound. Even if the performance of *C-SpriteOpt* is too optimistic as it currently focuses only on the hierarchical reasoning functionality, the lower bound performance set by *C-SpriteEsper* still clearly outperforms current approaches.

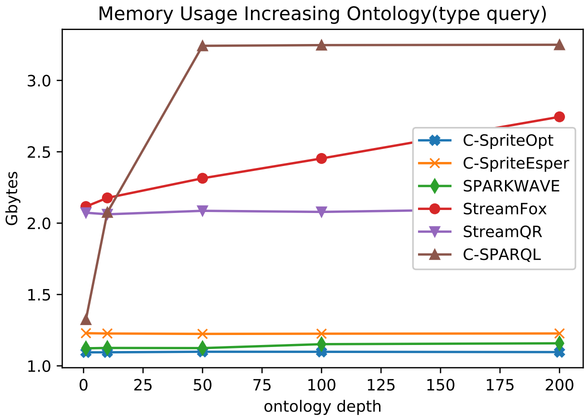


Figure 11: Evaluation of the memory consumption (less is better) of C-Sprite when increasing the ontology depth, compared to StreamQR, C-SPARQL, StreamFox and SparkWave.

9 DISCUSSION

In this section, we discuss how the approach is positioned in terms of traditional reasoning techniques and we discuss if the approach is feasible for non-streaming situations. As we stated in the introduction there are three approaches to perform reasoning. However, our approach does not naturally fit in any of those. We now discuss each of the approaches and how they relate to C-Sprite:

- Materialization: It is clear that C-Sprite does not perform materialization since it does not populate the Assertion Box (ABox) with new facts. However, it precomputes the hierarchy and when a new triple arrives, it links (in a memory efficient way) the triple to its parents. In this sense, C-Sprite is an efficient way of performing materialization.
- Query Rewriting: C-Sprite does not perform query rewriting since it does not rewrite any queries to contain Terminological Box (TBox) information. However, it builds a specialized data structure to create references between the queried types and the hierarchies. In this sense, C-Sprite rewrites the hierarchy in a specialized data structure.
- Goal driven: C-Sprite is not goal driven since it does not perform backward reasoning from the goals (the queried types) to the data. However, it maintains the relations between the queried types and the hierarchies in the underlying data structure. In this sense, C-Sprite proposes an efficient data structure that starts from the queried types (the goal) and goes back to the data.

It is clear that the C-Sprite does not fit into one specific category. We can conclude that it is a hybrid approach that allows to optimize the underlying data structures for a specific entailment that can be modeled in the form of hierarchies.

The power of the approach lies in the fact that we can check for each triple in the stream directly if it is needed for further processing. The question remains if this approach is feasible for non-streaming approaches. We argue that the approach is beneficial in situations where the data needs to be read from file, in this case we can process triple by triple and start answering the query while reading the data from file. In other cases, the approach is still feasible, but it will not result in the speed-up as in the streaming or reading from file cases. In Section 5, we formalized that the hierarchical reasoning can be performed before the rest of the processing, justifying that we can process triple by triple.

C-Sprite is an algorithm for efficient hierarchical reasoning, however, it can be more generally used by combining it with a query engine. C-Sprite can do the hierarchical reasoning, and filter unneeded triples while doing so, and the query engine can take care of the joins.

10 CONCLUSION

In this paper, we proposed an Stream Reasoning approach that exploits hierarchical language features from the underlying IFP. We have formalized the approach and shown in the evaluation that C-Sprite outperforms existing RSP engines for simple hierarchical reasoning tasks. We have focused on two types of triples only: class assertions and object property assertions. Furthermore, we formalized the approach for reasoning over the classes and did not take joining into account. We argue that the joins can be done at a later stage by utilizing, for example, a left-linear tree.

In future work, we wish to further formalize the approach, i.e. further generalize certain assumptions such as the sliding window of size 1 in Section 5. We also wish to exploit the hierarchy to enable more expressive reasoning.

Acknowledgment This research was funded by the VLAIO Strategic Fundamental Research (SBO) DiSSeCt.

REFERENCES

[1] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th WWW conference.* ACM, 635–644.

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

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer networks* 54, 15 (2010), 2787–2805.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web.* Springer, 722–735.

[5] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2010. Incremental Reasoning on Streams and Rich Background Knowledge. In *ESWC 2010, Proceedings, Part I.* 1–15.

[6] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2010. Querying RDF streams with C-SPARQL. *SIGMOD Record* 39, 1 (2010), 20–26. https://doi.org/10.1145/1860702.1860705

[7] Payam Barnaghi, Wei Wang, Cory Henson, and Kerry Taylor. 2012. Semantics for the Internet of Things: early progress and back to the future. *International Journal on Semantic Web and Information Systems (IJSWIS)* 8, 1 (2012), 1–21.

[8] Jean-Paul Calbimonte, Jose Mora, and Oscar Corcho. 2016. Query rewriting in RDF stream processing. In *International Semantic Web Conference.* Springer, 486–502.

[9] Olivier Cure, Hubert Naacke, Tendry Randriamalala, and Bernd Amann. 2015. LiteMat: a scalable, cost-efficient inference encoding scheme for large RDF graphs. In *Big Data, 2015.* IEEE, 1823–1830.

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

[11] Emanuele Della Valle and et al. 2016. Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In *Proceedings of DEBS.* ACM, 394–401.

[12] Daniele Dell'Aglio and Emanuele Della Valle. 2014. Incremental Reasoning on RDF Streams.

[13] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. 2014. RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. *Int. J. Semantic Web Inf. Syst.* 10, 4 (2014), 17–44. https://doi.org/10.4018/ijswis.2014100102

[14] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. 2017. Stream reasoning: A survey and outlook. *Data Science* 1, 1-2 (2017), 59–83. https://doi.org/10.3233/DS-170006

[15] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. 2011. A cep babelfish: Languages for complex event processing and querying surveyed. In *Reasoning in Event-Based Distributed Systems.* Springer, 47–70.

[16] Charles L Forgy. 1988. Rete: A fast algorithm for the many object pattern match problem. In *Readings in Artificial Intelligence and Databases.* Elsevier, 547–559.

[17] Srdjan Komazec, Davide Cerri, and Dieter Fensel. 2012. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of DEBS.* ACM, 58–68.

[18] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. 2011. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC.* Springer, 370–388.

[19] Alessandro Margara, Jacopo Urbani, Frank Van Harmelen, and Henri Bal. 2014. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web* 25 (2014), 24–44.

[20] Davide Martinenghi and Riccardo Torlone. 2014. Taxonomy-based relaxation of query answering in relational databases. *The VLDB Journal* 23, 5 (2014), 747–769.

[21] Mohamed Morsey, Jens Lehmann, Sören Auer, Claus Stadler, and Sebastian Hellmann. 2012. Dbpedia and the live extraction of structured data from wikipedia. *Program* 46, 2 (2012), 157–181.

[22] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A highly-scalable RDF store. In *ISWC.* Springer, 3–20.

[23] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. Context aware computing for the Internet of Things: A survey. *IEEE Communications Surveys & Tutorials* 16, 1 (2014), 414–454.

[24] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. 2005. G-ToPSS: fast filtering of graph-based metadata. In *Proceedings of the 14th international conference on World Wide Web.* ACM, 539–547.

[25] Xiangnan Ren and Olivier Curé. 2017. Strider: A hybrid adaptive distributed RDF stream processing engine. In *International Semantic Web Conference.* Springer, 559–576.

[26] Stuart J Russell and Peter Norvig. 2009. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,.

[27] Juan F Sequeda, Syed H Tirmizi, Oscar Corcho, and Daniel P Miranker. 2009. Direct mapping SQL databases to the semantic web: A survey. *Univeristy of Texas, Department of Computer Sciecnces Technical Report TR-09-04* (2009).

[28] Edward Thomas, Jeff Z. Pan, and Yuan Ren. 2010. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In *the Proc. of the Extended Semantic Web Conference (ESWC2010).*

[29] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. 2012. WebPIE: A web-scale parallel inference engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web* 10 (2012), 59–75.

[30] Raphael Volz, Steffen Staab, and Boris Motik. 2005. Incrementally maintaining materializations of ontologies stored in logic databases. In *Journal on Data Semantics II.* Springer, 1–34.