Stream Reasoning on Resource-Limited Devices

by

Colin Edward Hardy, B.A. (Mod.)

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

2013

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Colin Edward Hardy

24/08/13

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy

Signed: _____

Colin Edward Hardy

24/08/13

Acknowledgements

Firstly I would like to thank my supervisor Declan O'Sullivan and assistant supervisor Wei Tai, both gave much guidance in researching a topic as well as help in navigating the area of semantic technologies.

I would also like to thank my class mates for assistance in topics throughout the year in the face of a considerable work load, this guidance proved to be extremely valuable.

Colin Hardy

Stream Reasoning on Resource-Limited Devices

Colin Edward Hardy
University of Dublin, Trinity College, 2013

Supervisor: Prof. Declan O'Sullivan
Assistant Supervisor: Dr. Wei Tai

Abstract

Large amounts of data are now available on streams in areas such as sensor networks, social networks and financial markets. Much of the research on semantic reasoning so far however has been focused on static ontologies, but given the nature of the application areas, we wish to be able to undertake reasoning upon this stream information in light of complex background information.

In order to move away from centralised reasoning approaches we also wish to allow reasoning to be performed on resource-limited devices that could be potentially spread throughout the network. For instance if a phone is sending information to the central server it can perform reasoning and send the results as opposed to the raw data, this can help us to reduce the overall load on the server. Similarly for sensor networks, if the sensor can do some reasoning before sending on any data this could lead to an overall increase in efficiency for the network.

This dissertation looks at applying a recently developed technique for stream reasoning, but implementing it on a memory optimized reasoner. The key difference of the approach this paper takes however is that much of the streaming algorithm is handled at a lower level than previous approaches, this difference leads to cutting out a lot of unnecessary computation and an overall reduction in reasoning time for the reasoner.

There is a high variability in the data a reasoner can work over and the kind of environment a reasoner is in and so it can be hard to make overall comparisons between reasoners. In the evaluation a large number of experiments are presented in order to give an idea of how the reasoner reacts to different

scenarios, this includes an experiment comparing our reasoner with the reasoner on which it was based. It should be noted however that the original reasoner could only work for a single transitive predicate while our reasoner is capable of applying multiple rules to ontologies.

The reasoner was found to perform favourably when compared to the reasoner it was based upon and also constantly outperformed its naive implementation as well.

Contents

Figures

Chapter 1 Introduction

1.1 Motivation

There has been much research into semantic technologies ([30], [31] , [32], [33])for processing complexdata and harnessing these capabilities by allowing users to query over ontologies using particular query languages (e.g. SPARQL) to provide rich answers, but a lot of the research has looked at static ontologies where facts do not change. If we really want to make full use of ontologies in today's world there is a need to look at how to perform continuous complex reasoning in order to process constant streams of data in light of background information [3].

For example in [27] (a study undertaken on smart cities) the need for stream reasoning is evident when it is considered that much of the data necessary for smart cities will be high frequency data that is required to be processed quickly. Current semantic technologies have not been created with this kind of environment in mind.Other applicable areas include financial markets, social networks and sensor networks.

Reasoning can be a very taxing task in terms of memory [33]. While many methods focus on optimising reasoning algorithms or specific reasoner implementations, the idea of spreading reasoning through-out a network [14] is another potential approach that could lead us to more capable stream reasoning. Having small (in terms of memory) reasoners being able to perform some pre-processing on the data can help reduce the burden on a central server, leading to increases in performance.

Whereas some research has been undertaken on semantic stream reasoning and separately reasoners for resource limited devices, very little research has been undertaken upon semantic stream reasoning in resource limited settings [3].

1.2 ResearchQuestion

To what extent can an efficient stream reasoner for resource limited settings be implemented through alteration of an existing non-stream reasoner for resource limited settings?

1.3 TechnicalApproach

RETE level modifications along with lower level coding changes were required in order to add stream capabilities to Coror [12] (the name of the resource limited reasoner used in this research). The two metrics we were interested in measuring were speed of the reasoner and also the memory it consumes while undertaking stream reasoning. In order to give a sufficient idea of the reasoners capabilities a large set of ontologies, giving variability in size and expressivity were used in the evaluation. The window sizes of the stream were also varied in order to test the reasoner. A comparison with the naive approach to stream reasoning was also tested.

In the state of the art review, we examined the approaches people have taken in handling stream reasoning. Various different approaches have been taken but all take reasoning time as their main metric, with a few looking at memory consumption. Event processing was also reviewed, although it is not so relevant in the stream reasoning domain it still is applied to similar problems and may potentially give ideas on possible approaches.

After the state of the art review, it was decided that a rule based reasoner would be the best approach for this research, as there was much more information found on stream reasoning using a rule based approach. More specifically a RETE approach looks to create a RETE network at instantiation, followed by the network being populated with facts from the base ontology, once constructed there should be little processing required to do for subsequent processing, which is exactly the situation stream reasoning would benefit from. This compounded with the fact of having access to a reasoner designed for a resource limited environment led us to

adopting the approach of modifying a resource limited non-stream reasoner, named Coror, for stream reasoning purposes.

The Coror [12] resource limited reasoner is a RETE based semantic reasoner. It is a pure forward reasoner, that is it computes all consequences of facts at reasoning time, as opposed to leaving some to be computed at query time (backward reasoning).

The main features of Coror that differentiates it from usual RETE based reasoners are the fact that it shares alpha nodes between identical rule clauses, and also that it sorts the network in order to minimize the amount of partial joins that are fired that would result in no actual triples being created. Coror also has the feature of selective rule loading, i.e. it only loads in rules that will be fired. This can be important for reasoning as it helps minimise the size of our RETE network and not have unnecessary nodes in it, unfortunately though this last feature is not suited to wards stream reasoning. This is because we cannot be sure of what rules will be required as we do not know what information will be arriving via the stream. It should be noted that Coror is not a full reasoner, the rules Coror uses are given in the Appendix B.

A number of modifications to Coror were necessary to enable our stream reasoning approach. In order to make Coror capable of stream reasoning it must be able to associate timestamps with triples (temporal triples) and to then assign the correct timestamp to deductions which are caused by multiple facts (must have minimum timestamp of parents), also the RETE network must be able to store tokens with timestamps (temporal tokens). Once timestamps are assigned to both triples and tokens we can then remove the necessary items when their expiration time occurs. Coror must only contain the latest triples in its graph, so if a triple arrives into the ontology that the ontology already contains but has a greater timestamp then this must replace the original triple.

In the evaluation the metrics of speed and memory were measured. Various different ontologies were tested along with specifically looking at only varying ontology size and window size of the stream. A comparison between the stream reasoning approach taken in this work and the naive approach which involves re-

reasoning over the entire ontology+ triples in stream. Also an idea of how the reasoner created might compare with similar reasoners.

1.4 DissertationOverview

1.4.1 Background

The Background section looks at the technologies which our reasoner is built upon, specifically talking about the data language (OWL). Querying languages are spoken about to give the reader an idea of:

a) How information stored in ontologies can be harnessed and presented to users.

b) How stream information is extracted from an RDF triple stream.

Since our Reasoner is a pure RETE reasoner the RETE Network will be described at an algorithmic level, followed by a background of stream reasoning and the concepts that it is built upon.

1.4.2 SOA

The State of the Art looks at research into stream reasoning and the benefits and drawbacks of certain approaches along with research on reasoning on resource limited devices. Event processing is looked at briefly as it has a similar nature to stream reasoning but it should be noted that it is a separate field of research.

1.4.3 Design

The design section gives a high level description of the reasoners capabilities and also how they are achieved at an algorithmic level. The reasoner, stream processor and method for streaming are also discussed.

1.4.4 Implementation

For the implementation we speak about the specific changes and methods that were made/used in creating the reasoner.

1.4.5 Evaluation

In the evaluation all results that were collected are given and also discussed. Metrics measured were speed of reasoning time and also memory consumption

Chapter 2 Background

This chapter looks at familiarising the reader with the necessary background information required to understand the problems in stream reasoning. It gives a basis in semantic technologies followed by a quick overview of query languages used in ontologies. An outline of the RETE algorithm used for semantic reasoning is given also, this is the algorithm that is used in the implementation. Finally a background section on stream reasoning itself is given in order to underline the concerns of stream reasoning.

2.1 Introduction

In this chapter we shall give a brief introduction to sematic reasoners and stream reasoning. The chapter concludes with a look at reasoning and reasoning in resource limited devices.It should be noted that for much of the current state of the art research, memory is not taken into account.

2.2 OWL Reasoning

Reasoning is the process of going beyond the information given [15]. The ability togive machines certain facts and rules that then allow them to come to conclusions without the need for human interference is a powerful tool for complex systems. Semantic reasoners have then been created in order to process data sets and introduceexplicit logical deductions to these sets.

A lot of reasoners have a specified ontology language, usually defined generally by a frame language and then using a specific mark-up language (OWL, RDF,RDFS) to implement general concepts defined by the frame language. OWL, RDF and RDFS are based on description logic, they have been used in much research in recent times, being used regularly in the Semantic Web.

The W3C created the specification of Resource Description Framework (RDF) [16] to fulfil theneeds mentioned earlier; RDF was then built upon by RDF Schema (RDFS) [17] and Web Ontology Language (OWL) [1] to create a more complete model for structuring data in a way that could be reasoned over.

RDF structures data as by triples. Triples consist of a subject, object and predicate. The predicate is a relationship that relates the subject to the object. For instance if a person John has a dog we could create a statement to represent this information by using the subjectjohn, the predicate hasPet and the objectdog. OWL allows us to give the data implicit knowledge by giving us (for example) predicates such as transitive and inverse predicates, which reasoner can then use to come to new facts.

There are different sublanguages of OWL depending on how expressive a language the user wishes to use. OWL DL [19] has a strong relationship with description logic and is used by researchers who have developed reasoning algorithms using description logic.

Ontology Expressivity

Description logic has a classification system for describing how expressive an ontology is, the naming scheme is given below, a full explanation of the expressivity naming convention can be found at [36]. An example of a name would be ALCIN, ALC stands for attributive language, attributive languages properties are atomic negation, concept intersection, universal restrictions and limited existential quantification. C stands for complex concept negation, I for inverse properties and N is for cardinality restrictions. If an ontology is labelled as ALCIN then it must have all the previously mentioned properties.

Capabilities of OWL

OWL identifies ontologies and resources using URIs, it allows us to define property restrictions like setting the range of the property hasHusband to male. Also we can set cardinality restrictions, so say a child can only have two parents.

```
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">2</owl:maxCardinality>
</owl:Restriction>
```

(example taken from OWL RFC [29])

In similar ways inverse, symmetric and transitive properties (among others) can be created

Two popular categories for reasoning algorithms are rule-entailment [20] and Tableau based algorithms [21]. In Rule based reasoning a rule is an “if-then” operation, so if a certain condition is met, execute the related action.

Tableau reasoning uses a very different approach, it breaks down statements into their axioms by applying rules to them and consequently creates “tableau tables”, in Figure 2-1 there is an example of one of these tables taken from [25], the top statement is the original and the ones below are subsequent statements created from applying relevant expansion rules.

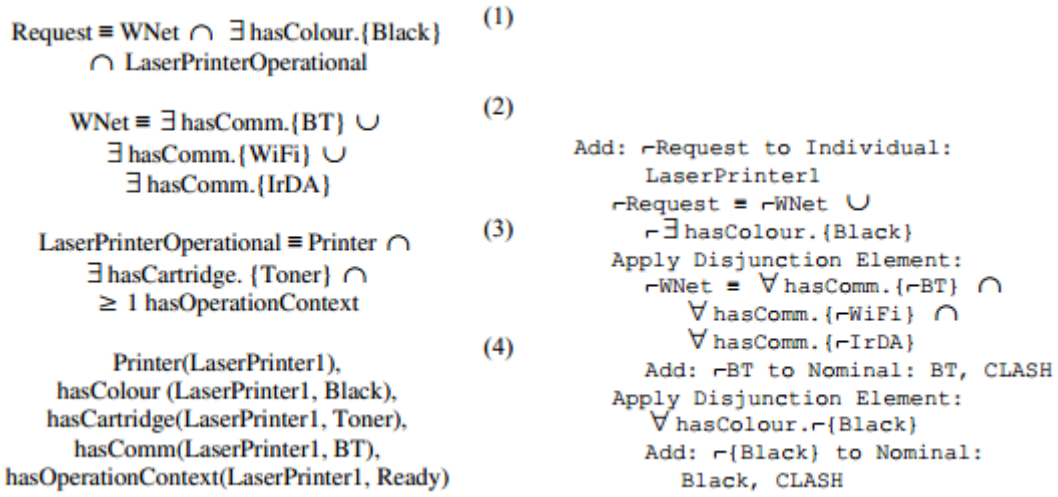
$$\neg((\forall x)(P(x) \vee Q(x)) \supset ((\exists x)P(x) \vee (\forall x)Q(x)))$$
$$(\forall x)(P(x) \vee Q(x))$$
$$\neg((\exists x)P(x) \vee (\forall x)Q(x))$$
$$\neg(\exists x)P(x)$$
$$\neg(\forall x)Q(x)$$
$$\neg Q(c)$$
$$\neg P(c)$$
$$P(c) \vee Q(c)$$

$$P(c) \qquad\qquad Q(c)$$

Figure 2-1 Tableau Expansion Table

If there are any contradictions in those axioms the statement is then considered un-satisfiable. When issuing a request using tableau, the negation of the request is given to the knowledge base, if there is a clash for every branch of the request then the relevant object is returned. An illustration of the reasoning process for a tableau reasoned was taken from [10] and is illustrated in Figure 2-2

Figure 2-2 Example of Tableau Reasoning



On the left is the definition of the request along with a relevant printer in the ontology, on the right is the processing of that request, finding a clash for every branch

A popular method for rule based reasoning is the RETE algorithm discussed in [7]. The basics of how the RETE algorithm works is that for each rule it creates a tree-like matching network (termed RETE network)that is separated into two networks, the alpha network and beta network. Each node of the Alpha network corresponds to the condition statement of a rule, with each node storing the facts that satisfy that rule. The alpha networks nodes filter facts while the Beta networks joins facts. Once a fact satisfies all the conditions for a relevant rule, the rule will be fired. The main strength of the RETE algorithm is that it does not need to iterate over the data itself as it uses filter mechanisms for which the data passes through. A more in-depth discussion of the RETE network will come later.

2.3 C-SPARQL/SPARQL

A key part of semantic technologies are the query languages, as without them any inferred relationships would become very difficult to extract, also it would make getting very specific information from the ontology very difficult, which would undermine a lot of the work done to ensure that ontologies have complex data

structure. SPARQL [38] gives us a rich query language which we can extract specific information from our ontologies.

While SPARQL is fit for its purpose it is not possible to use it over complex data streams. If we are provided with a stream which has a wide variety of information on it but we are only interested in a small subset of it then we require a way of querying these streams as well. This is where C-SPARQL comes in. C-SPARQL allows us to specify a window for triples in the stream, so anything received from the stream that is contained in the window will be returned. The window can either be a logical or physical window.Logical window means that all triples whose timestamps are within the current window will be returned, physical means that the last X number of triples which have arrived will be in the window. It then allows us to query over the triples contained in these windows so that we can retrieve the information we want. It should also be mentioned that it allows simultaneous querying from streams and RDF files which allows the user to easily merge a stream with a base ontology.

In Figure 2-3 there is an example of what a C-SPARQL looks like taken from [26]and how it can be leveraged:

```
REGISTER QUERY CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

SELECT DISTINCT ?district ?passages
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?tollgate c:placedIn ?street .
        ?district c:contains ?street . }
AGGREGATE ((?passages, COUNT, {?district}))
```

Figure 2-3 C-SPARQL Query

Here a query is made where we are looking for the number of passages made in a particular district by making use of where and aggregate clauses. Also the COMPUTED EVERY statement allows us to specify how regularly we wish to be returned results from the stream. The results that are returned could then be passed to a reasoner which could take into account some background knowledge in order to come to some new inferences.

2.4 RETE Network

A RETE network is an algorithm for implementing forward chaining semantic reasoning. Forward Chaining is a method whereby you incrementally use rules to then deduce new facts, and then use these new facts to deduce more facts, proceeding in this iterative fashion until no more new facts can be deduced. An alternative to this is Backward Chaining. Backward chaining is based on finding facts in order to satisfy a rule, it will start off with the conclusion of the rule and search for any criteria matching it. Backward chaining therefore requires a query in order for it to execute, example:

Advantages and disadvantages of one over the other are that forward chaining while being faster at query time consumes more memory due to the fact that all deductions are made at initialization, so every consequence is deduced whether it is needed or not. Backward chaining on the other hand is much more memory efficient as it will only make deductions necessary to answer the query, but on the downside it must search through the knowledge base and reason at query time, providing a slower response.

It is also possible to use both methods in combination with each other. These types of reasoners are commonly referred to as hybrid reasoners. The main idea behind them is to reduce the amount of possibilities the backward chaining engine much search through, mainly by the forward engine computing part of the deductions and then passing these results to the backward engine that can then finish off the reasoning

The Coror reasoner which our implementation is based around is a forward reasoner which uses the RETE algorithm.

The RETE algorithm is one the most popular algorithms for forward reasoning and is used by various semantic reasoners (at least partially). Itperforms semantic reasoning by first constructing a network, the network consists of Alpha, Beta, and Terminal Nodes.

The network is constructed by loading the rules which the semantic reasoner uses. Once the network is constructed triples are then fired into it which can then trigger deductions. A more detailed explanation is below.

Alpha Nodes:

Alpha Nodes are constructed by reading the “if” statements of the rules. So taking the example rule below we would get two alpha nodes given in Figure 2-4:

(?a type ?b),(?b subClassOf ?c)→(?a type ?c)

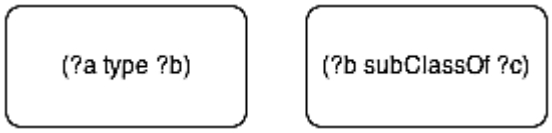


Figure 2-4 Alpha Nodes

When triples are fired into the network they will be checked with the above two alpha nodes if they match the pattern that is associated with the node, the node will pass it on to one of its continuations, i.e. the next node in the network. A continuation in this case would be a beta node, followed by a terminal node which would look like as follows:

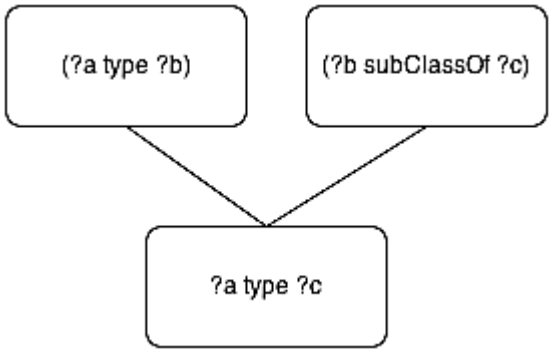


Figure 2-5 Alpha Node and Beta Node continuation

A beta node always has two input nodes which can either be alpha or beta nodes. Every time a triple arrives at a beta node the beta node will check for possible joins with that triple, it will join the triples and then send on the new triple to a terminal node which is then responsible for adding this new triple to our knowledge base.

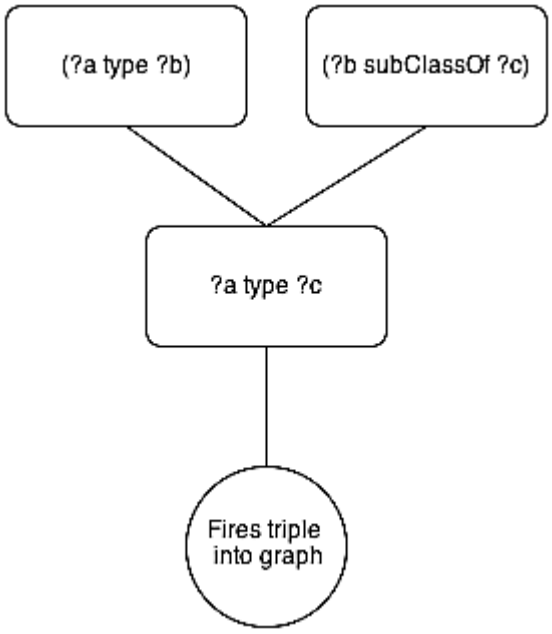


Figure 2-6 Alpha Nodes, Beta Node and Terminal Node

This process will be repeated for all triples, deduced triples are passed into the RETE network as well as triples that were contained in the original ontology.

It is not uncommon to have alpha nodes as the continuations of alpha nodes, filtering at each step. As a set of alpha nodes and their respective beta nodes are constructed for each rule there can sometimes be significant overlap between rule clauses, for instance thefollowing two rules:

(?a subClassOf ?b),(?x ?type ?a)→(?x type ?b)

(?a subClassOf ?b),(?b subClassOf ?c)→(?a subClassOf ?c)

Share an identical clause, rather than create separate nodes in the alpha network we can employ a technique called node sharing, Coror uses this technique in order to reduce memory consumption. Sharing alpha nodes then means we must be more careful in the implementation on how we decide to do joins.

It can be seen that the RETE network is suited to stream reasoning, as new triples can simply be passed through the above network.

2.5 Stream Reasoning

The aim of stream reasoning is to provide a means of reasoning over a continuous flow of data. While reasoning over a static knowledge base is a well understood problem the issue of having a knowledge base that is changing rapidly is not so well studied.

Applications of stream reasoning are large as there are now many devices that receive large amounts of information through streams, but are not harnessing it. Examples include traffic monitoring, financial transaction auditing, wind powerplant monitoring, situation-aware mobile services and patient monitoringsystems[2].

When we are dealing with data coming in on streams we must be able to have some range over the stream which we can extract in order for it to be of any value to us, as we cannot simply store all the information that arrives on the stream, since for fast complex systems this would overload our system easily, at the other extreme we then have only storing one item at a time, with no context with the rest of the stream this is not ideal. Neither of these approaches are therefore viable. A popular approach to stream reasoning is then a window based approach[3], this means we set an interval over the stream in which we consider those items in the stream valid.

Thiswindow based method for stream reasoning can be then seen to separate the knowledge base into dynamic and static parts, (the dynamic part is the set of triples contained in the streaming window) and to then assign any incoming triples on the stream timestamps, the window then defines what range of timestamps should be considered when reasoning. The simplest way of implementing stream reasoning would be then to just reason over the entire available knowledge base again once the window updates, however this is highly inefficient [4]. Methods for window based reasoning are discussed in the state of the art chapter.

2.6 Summary

Now that the reader has had an introduction to the necessary topics, the state of the art will follow in order to take an in-depth look at approaches taken to stream reasoning and try to evaluate the differing approaches pros, cons and scope.

Chapter 3 State of the Art

A number of different approaches to reasoning were studied, below is a breakdown of the titles of the approaches, along with whether their approach took into account time of reasoning, memory consumption and also whether or not the reasoner supported stream reasoning. The last column is the methodology used by the reasoner.

This chapter is broken up into 3 main sections. The first section covers research concerned with stream reasoning, the second looks at event processing in order to see if there is any potential use of ideas in this domain due to there being some similarity between stream reasoning and event processing. The third section then looks at reasoning on resource limited devices.

Table 3-1 shows all the approaches that have been researched in this chapter. It gives the metrics with which the research was concerned, and whether or not this approach concerns itself with stream reasoning.

Table 3-1 Table of Differing Reasoning Approaches

Approach	Time	Memory	Stream Reasoning
Dred Method [8]	yes	no	yes
Incremental timestamps [5]	yes	no	yes
Truth maintenance system [13]	yes	yes	yes
mTableuax [11]	yes	yes	no
Selective Rule loading [12]	yes	yes	no
2-phase RETE [12]	yes	yes	no
Distributed Tableau [25]	yes	yes	no
Syntactic Approximation [18]	yes	no	yes

3.1 Stream Reasoning

Incremental reasoning's [4] aim is to use previous conclusions made by the Reasonerto prevent unnecessary re-processing of certain facts and rules. In Figure 3-1 the concept of incremental reasoning is outlined.

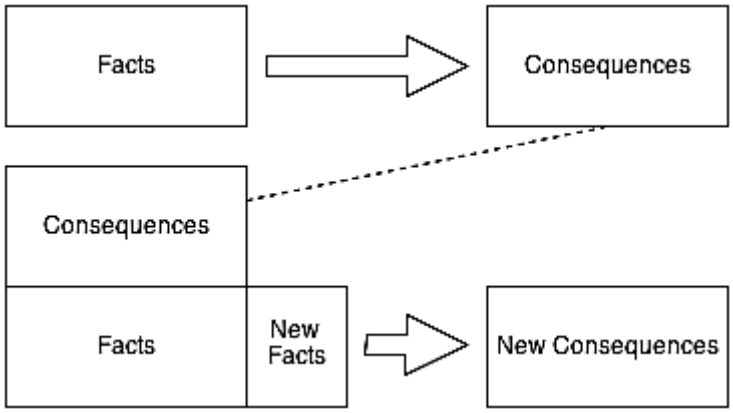


Figure 3-1 Incremental Reasoning Diagram

This works very well for monotonic reasoning [4] where facts cannot be removed, however for non-monotonic reasoning where facts can be removed with time, as opposed to just being added, problems of consistency and removal of facts become an issue. For instance in the Figure 3-2:

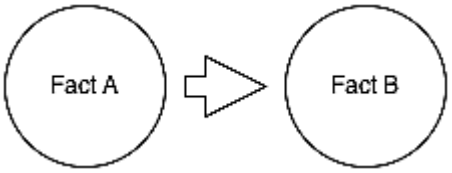


Figure 3-2 Fact Entailment

If Fact A were to be removed from the ontology how could we know to remove Fact B? This is an important problem to solve, differing approaches are highlighted in the coming section.

Another problem for Stream Reasoning illustrated in [14] is that of being able to handle highly dynamic data, in that it has a tendency to back-up if there is complex reasoning being done with that data, due to length of time it takes to reason over the data compared to the frequency at which the reasoned receives it.

This can then lead to a snowball effect which will in all likelihood leave cause the reasoner to be overwhelmed.

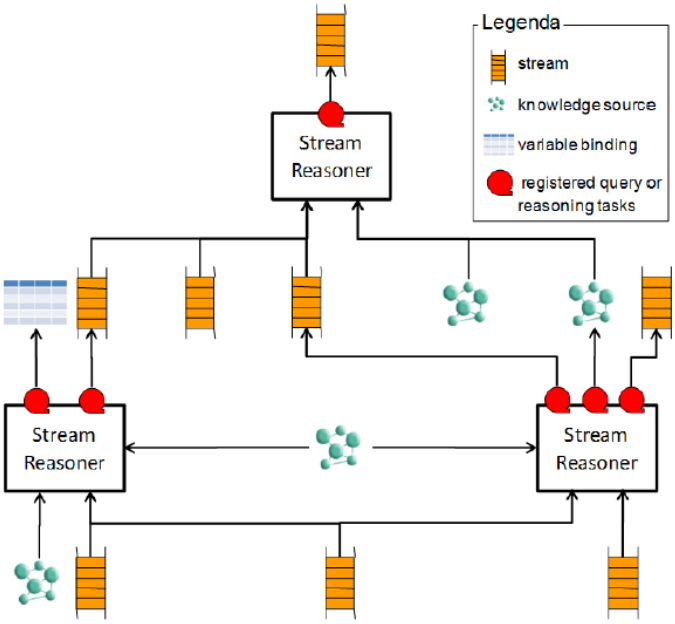


Figure 3-3 Distributed Reasoning Network

A solution proposed [14] that is illustrated in Figure 3-3, is to create a network of reasoners, having simple reasoners at the start of the reasoning process (e.g. filtering data) and then at the end having more complex reasoners, ideally then any data that reaches these complex reasoners would have dramatically decreased in frequency and would therefore be more manageable. While this is a possible approach it must be noted that the applications of this sort of architecture will be more limited than a single reasoner since not all scenarios may be capable of having multiple reasoners interacting with each other.

A suggestion to solve the problem of consequences of facts not being removed[4] is assigning timestamps to entailments and storing intermediate conclusions to prevent re-processing. This method was tested [5] and compared with a naïve implementation, while the time taken for re-reasoning was lowerfor cases where the knowledge base changed between 0-13%. A second method that was also compared in [5] was the delete re-derive (DRed [8]) method. It works by firstly removing facts and all their consequences, it thenadds any new facts andcomputestheconsequences of these facts along with re-computing any facts

that were deleted but had other possible derivations. The method is not as fast as the entailment timestamp approach, however it still out performs the naïve approach for cases where the knowledge base changes between 0-2%. The DRed method was introduced in [8], it also comments on a potential method to increase optimisation by separating rules into constituent parts, and then using the DRed algorithm on these to help reduce the amount of un-necessary deletion and re-derivation of entailed facts. While this may help increase performance it will most likely be at the cost of memory.

While the DRed algorithm has gained a lot of interest it is quite inefficient has many of its deletes and re-derivations will be a complete waste in scenarios where certain consequences and facts have not in fact changed.

It should be noted that both of the above approaches in [5] compared were implemented using the Jena rule engine. All the extra functionality of updating timestamps were created asan add-on by using Jena custom built-in functions. Also the reasoner was only designed to work for a transitive isIn property, which requires a large table of rules to be constructed for this particular predicate. In a situation that requires full owl reasoning, there would need to be a way of dynamically creating these tables in order to be able to process custom ontologies, also this would lead to a large amount of tables being created in some cases which would consume more memory on the reasoners part.

The rule table for the timestamp entailment approach is in Figure 3-4:

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

Figure 3-4Timestamp Entailment Rule Table for Stream Reasoning

Therefore the above cannot be thought of as “real” reasoners but are just used as a proof of concept.

A drawback of the timestamp entailment approach is that one can only use a static window for reasoning, having a dynamic window may be useful in situations where the relevance of previous data may fluctuate. Another short coming of the above approach is that there is no way to delete assertions in real time, such capabilities are desirable in situations when you have multiple input streams which may introduce inconsistencies. A method [13] to solve this problem was proposed which involves the use of truth maintenance systems. Truth maintenance systems store all intermediate results and also capture relationships between facts and their consequences, when a fact is removed then all of the previous consequences of this fact can be removed easily.However storing the intermediate results in the system obviously take up large amounts of memory, the paper then goes on to discuss optimizations in order to reduce memory consumption. The optimisation involves the use of backward-chaining in order to determine what intermediate results will be used,and if they are not going to be used they therefore will not be stored.

This optimisation is interesting however is not as suited to stream reasoning particularly, since it if a new triples were to arrive which would cause a previously deemed “un-used” triple to be used then potentially this could lead to a slower reasoning time for incoming triples.

Backward chaining is a way of determining whether a particular individual satisfies a rule, it is termed backward chaining as it begins with the goal of the reasoning process. For example in reasoning who is a student, the program would start with “X is a student”, where X is the set of all individuals that are students and is currently unknown, it would then consult the rule base to find the rule “if person attends college or school then person is a student”, to determine that X is a set of people who satisfy the above rule and would then find all individuals that satisfy this rule. The optimised Truth maintenance systemand non-optimised Truth maintenance system were then compared with the optimisation working favourably in terms of processing time, no figures were given for memory consumption, and instead stored intermediate triples were shown. They then compared the optimised TMS with a naive implementation, the naive implementation was faster to initialize due to the fact that it does not need to store intermediate results however it was consistently much slower for performing updates on the ontology.

Another approach[18] for solving problems with traceability of consequences also concerns itself with storing traceability information on derived facts and deriving facts. As mentioned previously complex data can slow down a reasoner causing it to fall behind and become overwhelmed.Syntactic approximation of underlying OWL languages is an approach to simplify OWL-DL expressivity, in order for there to be less strain put on the reasoner. The paper approximates ontology expressiveness to EL++ which allows for use of its tractable incremental reasoning facility [22], the approximation is then extended allowing for the creation of graphs in order to follow consequences of facts.

The above approach was compared with the naive implementation and was found to outperform it significantly with respect to speed. Future extensions of the work involve more intelligent tracing, so that certain consequences which don’t need to be deleted are not and also maintaining erased information that might need to be re-reasoned soon after, in other words having a sort of a buffer for deleting consequences. Both of these future extensions would most likely come at the cost of memory consumption.

3.2 Event Processing

Event processing is an area of study which has a similar grounding as stream reasoning, although it should be notes that it is a separate field of study. Event processing is concerned highlighting significant events. However by itself it does not take into account any sort of background knowledge on the events and it also cannot perform any sort of reasoning tasks, only generally able to take into account temporal displacement of events. Naturally people have then tried to extend this model to one which can make use of background knowledge [6], leading to a mentionable overlap between the two areas. One such implementation is ETALIS[9]. ETALIS is an event processor, which can take into account background knowledge and also more complex temporal relationships in order to help report more complex events. EP-SPARQL [23] has then been developed to make use of event processors such as ETALIS in stream reasoning. By being able to query the stream which then returns triples with timestamps.

While event processing is a notable mention in this paper and could be a useful tool for stream reasoning, it is not the focus, as event processing does not itself have to deal with the problems of consistency and efficiency that are the main hurdles to overcome in stream reasoning.

3.3 Reasoning on Resource Limited Devices

All of the approaches above do not take into account memory considerations for reasoning; they are generally only concerned with speed.

It has been highlighted in [11] that there is room for optimization for memory using the tableau algorithm, this is due to the fact that the algorithm looks for clashes, and there may be rules more likely to produce clashes than others, in which case they should be tested first, also the whole ontology does not need to be loaded for inference checks, in which case we can save memory.

An optimization of the tableau reasoning algorithm that also accounts for memory concerns is proposed in [10]. The method involves introducing weights to objects and disjunctions, and then ranking these based on the weights, consistency rules

will then only be applied for certain weights in order to decrease memory consumption. Skipping disjunctions and consistency rules resulted in the best way to optimize the approach in the paper, introducing weights for objects gives more overhead and as such is not as desirable. Classical tableau reasoners would give out of memory results if the reasonerwere to run out of memory, future reasoners are being proposed to do partial reasoning and return a result but with a probability that the result is wrong.

mTableaux [11] was created with the basis of [10] and was found to outperform most reasoners in a non-memory constrained setting, and out-performed all reasoners in a memory constrained setting, figure 3-5 shows all the different tests that were done with the reasoner, followed by Figure 3-6 detailing the results.

Test #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Selective Consistency	×	×	×	×	×	×		×	×						
Rank Disjunctions			×	×				×	×	×		×			×
Rank Individuals		×	×			×			×	×			×	×	×
Skip Disjunctions	×	×	×	×			×			×				×	

Figure 3-5 mTableaux test table

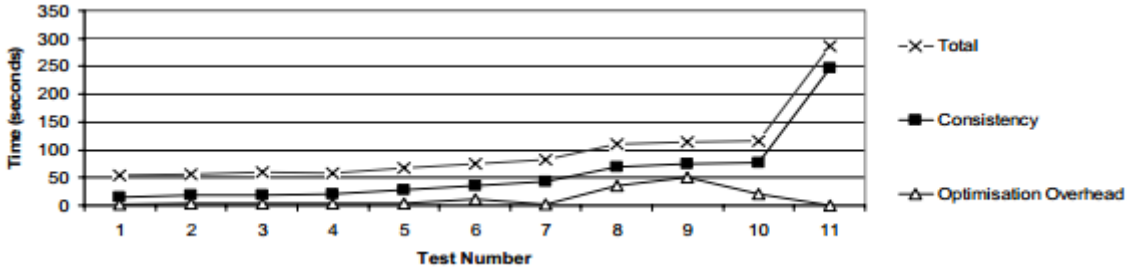


Figure 3-6 mTableaux Results Graph

Methods proposed [12] for reducing resources used by reasoners. Particularly it offers two novel approaches using reasoned composition methods, where composition means that the Reasoner makes relevant changes to how it will reason over an ontology depending on the way that ontology is structured. The first method is a selective rule loading algorithm. It works by examining the rule set and ontology and observing whether or not a rule will ever be required to be fired, if the answer to this is no then the rule will not be loaded and the reasoned will then save memory.

The second approach involves is described as a two phase RETE algorithm. In the first phase it allows the sharing of alpha nodes in the network for similar rules, this helps cut down on memory consumption. Then statistics are gathered on how many facts satisfy certain rules, the second phase begins and beta nodes are constructed by starting with the most specific rules. By starting with the most specific rules and ending with the least specific rules we are able to cut down how many intermediate results must be stored and reduce overall memory consumption. The implementations were then tested and were found to greatly reduce memory consumption and time spent processing on the testing device.

One interesting approach with regards to memory constrained reasoning using a tableau approach in the case of modular ontologies(that is to create an ontology by combining several smaller ontologies) is to split the reasoning among reasoners dependent on domain [25]. The reasoners then communicate with each other when they require certain statements to be branched which are not in their domain and exchange results, determining whether an ontology is inconsistent or not. This approach is more memory efficient as when new information is introduced each reasoned only needs to determine effectsin their own table, and therefore certain relations will not need to be calculated.

Although mTableau claims to be made use of in mobile services it does not outline a method for processing stream information.

An important note to make with regards to memory constrained devices is that optimizing to minimize memory consumption is not in itself enough to ensure resource limited reasoning. While having algorithms optimized for memory consumption allows a reasoned to reason more effectively the problem still remains where what if the devices memory is still all consumed? Mixing probability with logical statements has been studied in the past [25].

The mTableau [11] implementation mentioned several ways of improving their implementation in a resource constrained setting. Notable methods they mention are ranking requests such that more important reasoning will be done first and will therefore be more likely to return a result. Another is since mTableau already has weights for disjunctions and terms, it can create a cut off such that certain facts won’t be reasoned if they are below it, the cut off can be changed when memory is

beginning to run low. The final suggestion they make is to have a selective rule loading algorithm as well, similar to what was seen in [12].

3.4 State of the Art Recap

In this chapter,we have gone through a number of different approaches and implementations in stream reasoning each with their relative pros and cons, while there is a lot of work connected to stream reasoning little of it is concerned with memory limited devices, with a few showing concern for memory in general. Those that do so far have only optimised for memory consumption instead of trying to find a way to prevent out of memory errors, which is a subtle distinction but an important one.

Chapter 4 Design

This chapter gives a high level description of the reasoner, detailing the components used and also the requirements for stream reasoning. It also gives a description of the stream generator that was used in order to pass triples to the reasoner.

4.1 Introduction

The stream reasoner was created by building upon a reasoner developed in TCD called Coror [12]. In this section we will split design into 3 sections. The first will give a high level view of the main components used in our stream reasoning implementation, the second will speak of Coror in order to give the reader enough information on the reasoner in order to understand the necessary modifications made to allow for stream reasoning. The 3rd and final section will then speak of the modifications made. Specific details will be contained in the implementation section.

In the coming sections, when a fact is stored in a graph, we refer to it as a triple. However when a fact is stored in the RETE network, it will be referred to as a token.

Figure 4-1 gives the main components of the design:

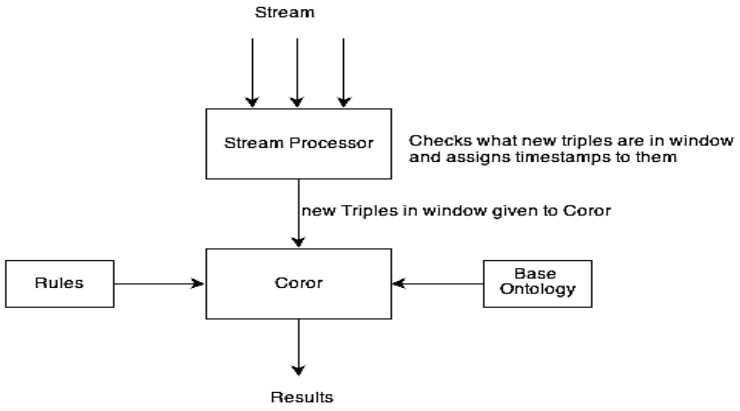


Figure 4-1 Stream Reasoner Design

4.2 Overview

Incremental reasoning, which has been mentioned briefly in the state of the art, is method for reasoning whereby the reasoner re-uses previously deduced facts when it needs to reason over similar data multiple times. The benefit of this approach is that it reduces reasoning time as it can re-use old results. This approach does not bring up any complications in the case of monotonic reasoning where the fact base can only increase. However as stream reasoning is designed around the idea of triples being able to expire, there needs to be a safe way to remove a triples consequences.

The design of the reasoner can be thought of has having two separate components. The first is the window processor, which allows the RDF stream to be queried and returns a specific set of triples, the set of triples will depend on the query given to the engine. The triples supplied to the reasoner in the window will be assigned timestamps which indicate when this particular triple should expire and be removed from the graph. The reasoner uses background information which is static on the reasoner, and then combines this information with the time stamped triples introduced from the window.

Axioms can fit into two categories, TBox (Terms) or Abox (Assertions) axioms. Tbox axioms contain facts such as (class1 subClassOf class2) while Abox axioms contain references to individuals (John type Person). The facts which are streamed are all Abox axioms, this is due to the fact that Tbox axioms are applied to ABox axioms, which allows new deductions to be made about these ABox axioms. For this reason all triples in the stream are ABox axioms in our implementation, although there is no technical problem with streaming TBox axioms. It also allows us to use popular ontologies more simply, as we are not forced to change any of the relationships inside the ontology.

There were infact two separate stream querying components. This was requiredas the memory usage of the C-SPPARQL library is very high, especially when one considers that much of its functionality(very expressive query language) is not being used by the implementation.

The second component is the reasoner itself, which can take in background information as well as the current window of the stream and perform reasoning on this data combined.

The third section discusses the modifications required in order for Coror to be able to perform stream reasoning correctly. The basic design of the stream reasoning was taken from [5], specifically the timestamp entailment approach, although we will have to specify more requirements as our implementation occurs at a lower level than [5].

4.3 Stream Window

For a stream every single triple that is received cannot be held forever but also it is nearly useless to only store one triple at a time in memory. The stream window is an interval which allows the taking of a set of triples from the stream and to deem them relevant. The idea of this is that information which comes in from a stream has a period of time in which it is relevant (i.e. when it's in the window), and then at some point in time this information in the window becomes irrelevant and expires. These two ideas are then encapsulated in the form of the stream window [3].

While in this work there are two implementations of the stream window they both come from the same design concept. Triples in the stream will have a timestamp assigned to them by the time they arrive at the reasoner.

(<Subject, Predicate, Object>, Timestamp)

C-SPARQL allows us to specify whether we want the window to be one specified by time i.e. take all triples whose timestamp are within the interval [0,2]. Below would then be the criteria a triples timestamp must pass if it is to be included in the current window, with a window size of 5 seconds say.

time-5<Timestamp<time

We are then allowed to specify how regularly we wish for the stream to be queried.

We may also just select the last x triples which have arrived into the window. Regardless, in the end the results our reasoner is given are a set of triples.

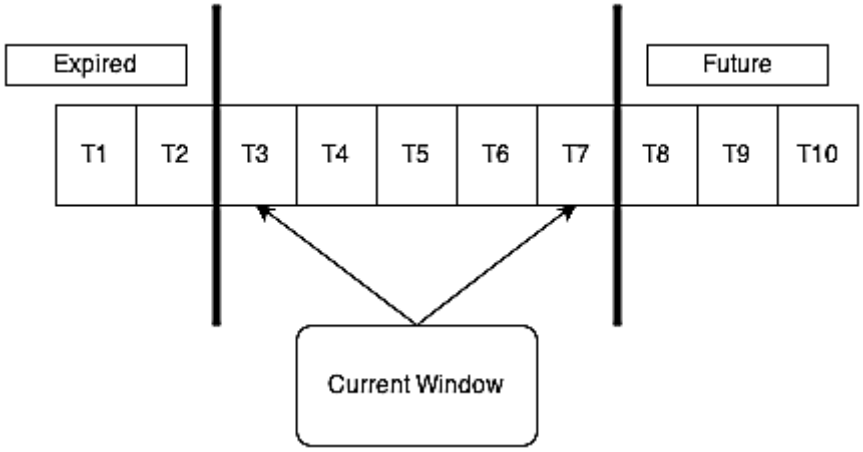


Figure 4-2 Stream Reasoning Window

One of the big advantages of using a window based approach is that if the windows are overlapping, so say we have a window of range 5s but increase the step by 1s every iteration, which means we have 4s of overlap between each window. While the stream processor will just return all the triples in the current window, the reasoner will be able to detect which triples are overlapping and will therefore not re-add un-necessary triples, leading to a much more efficient reasoner in cases where the window is very large compared to the step size.

4.4 Coror

Coror is an implementation of an optimized forward chaining RETE reasoner. The main concepts have been discussed in the state of the art. The two main optimizations Coror introduces are:

1. Selective Rule Loading Algorithm
   Simply put the reasoner will only load rules that are going to be fired. This is done by checking that there are triples in the ontology that will match the clause indexes of the rule.

   As mentioned earlier, this rule loading feature however was kept off for any sort of stream reasoning, as mentioned before it is a similar concept to the optimised truth maintenance systems mentioned in the SOA. Since we do not know what kinds of triples will be arriving on the stream, we cannot say for sure what rules will and will not be fired.

2. Two Phase RETE Algorithm
   The first phase of the two phase rete algorithm involves the use of alpha node sharing. That is when rules contain a common clause that these rules will then share this alpha node.

   The second phase is aimed at minimizing the amount of intermediate joins done by a rule. For rules that contain a large number of clauses several intermediate joins may need to be done before it is decided whether the rule is satisfiable. For instance if a rule has 3 clauses, the network will be required to do two joins to fire the rule, this is attributed to the fact that a single beta node takes two input nodes. This means that our network may do multiple intermediate joins but at the end discover that the rule will not be fired.

   For this reason Coror make sure the most specific joins are done first, this reduces the chance of doing many intermediate joins without coming to the rule firing. How specific a clause is then determined at load time. The

clause with the most triples in a rule will be considered the least specific and the clause with the least will be considered the most specific.

Below Figure 4-3 gives lists of tokens that are about to be joined

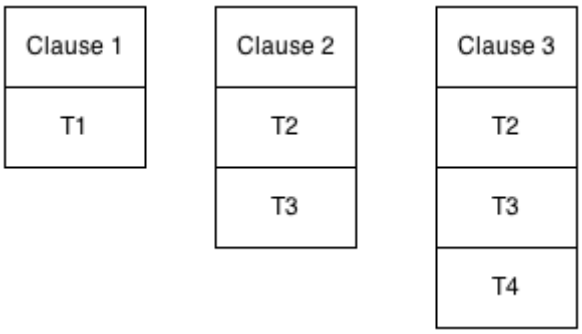


Figure 4-3 Tokens in RETE network that will be joined

If we are to join clause 1 and clause 2, then join the result with 3 we will minimise the amount of unnecessary joins, discovering immediately that there are no common terms to join at the first step. However if we were to join clause 3 and clause 2 and join this result with clause 1, we will join T2 and T3 only to then find that there is nothing in clause 1 to join this result to. This is the reasoning behind the second phase.

4.5 Modifications for Stream Reasoning

In order to make Coror capable of stream reasoning we must have principles for updating and assigning timestamps to triples. Our main design as mentioned earlier is taken from [5].

Its three main design points for the reasoner in [5] were:

1. Computes the entailments derived by the inserts,

2. Annotate each entailed triple with an expiration time

3. Eliminates from the current state all copies of derived triples except the one which has the highest timestamp.

As the implementation for [5] was undertaken using Jena's generic rule engine [35] and was therefore not handling reasoning at the engine level, the design above is missing a few extra requirements. As there must be criteria for facts that are also combined, since [5] only deals with a simple transitive property, and sothe following requirements must be introduced in addition.

4. Alltokens placed in RETE network must be removed at expiration time.

5. Products of joins between two temporal tokens must output a temporal token with the minimum timestamp of its two parents. The product of a triple with a temporal triple must output a product with timestamp of its temporal parent.

6. Facts contained in the base ontology should not be replaced by temporal versions of themselves

7. Tokens, similar to triples in the ontology, must be updated in the case that of duplicate tokens arriving into the RETE network if we are to preserve correctness in our joins.

4.5.1 Explanation of Requirements

4.5.1.1 Requirement 1

The first requirement is a basic reasoning requirement, our reasoner must be able to take a set of facts be able to deduce the appropriate entailments.

4.5.1.2 Requirement 2

By labelling entailments with the minimum timestamp of the facts used to deduce them, triples can be safely removed from the graph based on their timestamps without having to worry about figuring out their consequences and seeing if they should be removed. Figure 4-4 illustrates this approach.

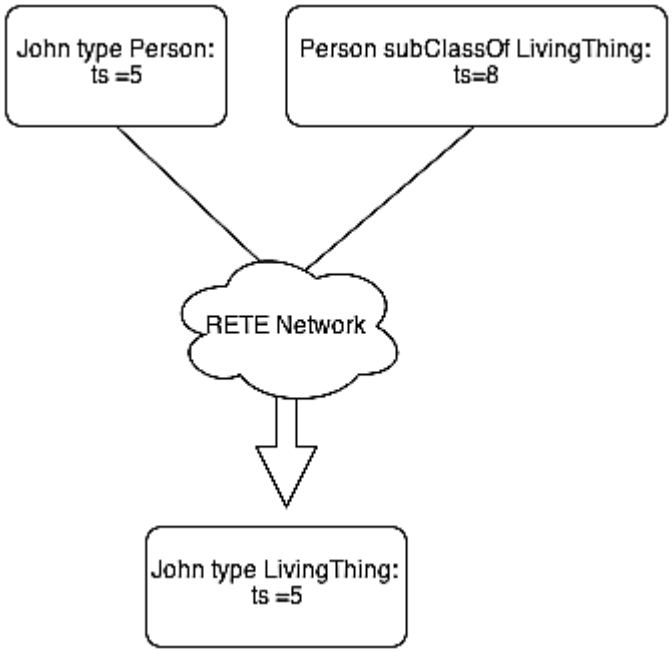


Figure 4-4 Example of Timestamp Entailment

When the current time reaches 5, the triples (John, type, Person) and (John,type,LivingThing) will both be removed.

4.5.1.3 Requirement 3

Requirement 3 stops the reasoner from having multiple duplicates of triples with different timestamps, if left in this would lead to large amounts of unnecessary computation being done.

4.5.1.4 Requirement 4

Requirement 4 is introduced in order to prevent any unwanted joins being performed with past and present triples and to also prevent large amounts of memory being consumed over time. This is required due to the extra level of abstraction in our work.

4.5.1.5 Requirement 5

Requirement 5 is an intuitive step that is needed for rules which are joined from multiple facts that may expire. If there is a conclusion Z that has been derived from facts X and Y, then it is necessary that's Zs expiration time should be the minimum of x and y, as once one of the facts that is necessary for Z to exist expires, then Z should expire as well.

4.5.1.6 Requirement 6

Requirement 6 may not be entirely necessary in some situations, however, it does lead to significant performance increases along with being an intuitive act. Step 6 means that if a fact is contained in the original ontology, but is then also deduced later on from some of the temporal triples which arrive in the stream, then the original fact will not be replaced. The reasoning behind this is due to the fact that if a fact does not have a timestamp originally then it is persistent data and should not be removed. This can then lead to significant performance gains, as if an underlying fact IS changes to a temporal fact, then all of the conclusions that were derived from it will now also become temporal. This can sometimes lead to large amounts of an ontology then becoming temporal, which can greatly increase the overhead of the reasoning process due to the steps above.

4.5.1.7 Requirement 7

Old tokens must be updated in the RETE network in the case of a duplicate token with a higher timestamp arrives.

4.6 Stream Generator Design

The key design requirement for the stream source was to have triples generated and put into a stream for the stream processor to then receive. The stream generator continually generates triples, and has different modes in order to accommodate different ontologies. It is important for the evaluation that the set of triples being streamed for ontology must invoke full expressivity of the base ontologies. For instance if an Ontology expressivity claims to have inverse relations, if none of the triples streamed into the reasoner invoke this inverse relation then any deduction we make about the reasoning time due to the expressiveness will not be valid. While most of the triples generated for the ontologies are instances of classes contained in the ontology, the LUBM ontology has had a data generator [37] developed for it.This generated data is then used in all streaming experiments with the LUBM ontology.

4.7 Summary

Now that the reader has a high level description of the reasoner, the following chapter will focus on describing the implementation, where the modifications necessary to support stream reasoning will be discussed at a lower level. Going into details on specific changes needed to be made to the RETE network etc.

Chapter 5 Implementation

5.1 Overview

The implementation of the Stream Reasoner is in Java. The implementation can be separated into the following components.

- Streaming
- Stream Window Processing
- Stream Coror
    - Graph Level Implementation
    - RETE Level Implementation

Streaming is how we create theRDF stream which is passed to the stream processor. For our implementation, sockets were used in order to provide a means of communication between the reasoner and the source of the stream.

Stream Window Processing is involved in the capture of data from the stream and how it then introduces this captured data to the reasoner.

Graph Level Implementation can be thought of how the triples are added to the ontology. It is separate to the RETE Network, the interactions between the two is the RETE network has triples passed into it from the graph and then the RETE network then passes out a series of consequences to the graph.

RETE Level Implementation is then involved with constructing the RETE network correctly and also being able to maintain it sufficiently by removing/adding tokens.

We will also speak about the interaction of Coror and the Stream processor during reasoning.

Class diagrams for the main Coror class, the RETERuleInfGraph (the graph used to sweep and add triples to coror) and also the RETEEngine are included in **Appendix C**.

The interactions of Coror and the Stream processor during reasoning are discussed once the reader is given a grounding in the two separate components.

5.2 Streaming

The data stream was implemented using sockets. It had various modes in order to work with different ontologies. The data is then received at our stream processor which extracts the current window of triples. Triples are inbuilt into the stream source, sending instances of certain classes to the reasoner. For the LUBM ontology, the triples sent are generated using LUBMs ontologies own data generator.

5.3 Stream Processing

There are in fact two separate implementation created for the stream processing implementation, as mentioned earlier this was due to the high memory usage of the C-SPARQL library. Both implementations will be discussed as they each have their own strengths and weaknesses. The non-CSPARQL implementation shall be referred to as the Bare Implementation from here on in.

5.4 C-SPARQLStream Processing

C-SPARQL is a modification of the SPARQL query language that allows the user to query RDF streams, the library gives the ability to assign an observer to an RDF Stream andthis observer then returns results periodically which we can be passed to the reasoner. It allows the user to give queries where theycan specify the range of the window and what triples theywish to be returned. Below is the query used to return all triples in an RDF stream which are currently in the window.

```
"REGISTER QUERY StreamQuery COMPUTED EVERY 1s AS "
        + "PREFIX ex: <http://localhost/default/tr069#> "
        + "PREFIX f: <http://larkc.eu/csparql/sparql/jena/ext#> "
        + "SELECT ?s ?p ?o (f:timestamp(?s,?p,?o) AS ?ts)"
        + "FROM STREAM <http://localhost/default/tr069> [ RANGE 20s STEP 1s] "
        + "WHERE { ?s ?p ?o }"
```

It returns the subject predicate and object of triples in the current window similar to a normal SPARQL query. However it also returns the timestamps of these triples as well which is a unique feature of C-SPARQL. In cases where the stream is very complex and may contain large amounts of information, C-SPARQL allows for the queries to berefined through use of aggregate and group built-ins.

It should be notes that the timestamp the triples have on the stream are in fact different to this timestamps that the triples have when they enter the reasoner, this will be explained in more detail soon.

In the implementation that uses the C-SPARQL library a tumbling window query was used, this means that we can be given all the triples that arrive in a 1 second interval. This is then passed to the reasoner.

Example of Processing for a 5s window size:

When the reasoner is started the stream processor will query the first 1 second interval at time=0s, all triples in this interval will be passed to the reasoner with expiration time 0+5=5s, the second interval will be queried at time=1s and all triples in this interval will be given a timestamp of 1+5=6s. Eventually when the reasoner reaches time 6 the initial triples that were introduced with timestamp 5s will be removed along with any expired tokens in the RETE network. This also provides an effective way of making sure the latest triples contained in a window will be in the ontology, as the implementation can handle it all at graph level operations and does not need to account for the specialized case of when duplicate triples are in the same window.

Figure 5-1 is a diagram taken from [28] that shows the C-SPARQL query Engine.

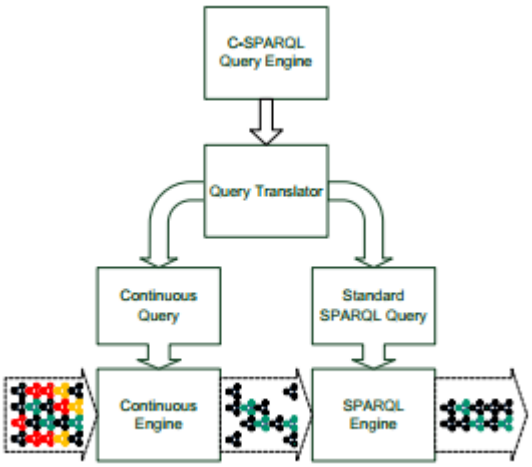


Figure 5-1 C-SPARQL Overview

A full class diagram of the C-SPARQl library can be found in Appendix A.

An Engine is first created at which point the user can assign an RDF stream (all the triples we are streaming to be queried, and also assign an observer, which gives access to the triples that are captured each time the query is computed.

5.5 Bare Stream Processing

The second implementation for stream processing was done from scratch. While the processor has much less functionality than the C-SPARQL implementation, it is still able to select triples from the current stream window and present them to the reasoner. It consumes much less memory than the C-SPARQL implementation and also is able to be run on the Android platform.

Figure 5-2 is theclass diagram for the stream processor given pm the next page.

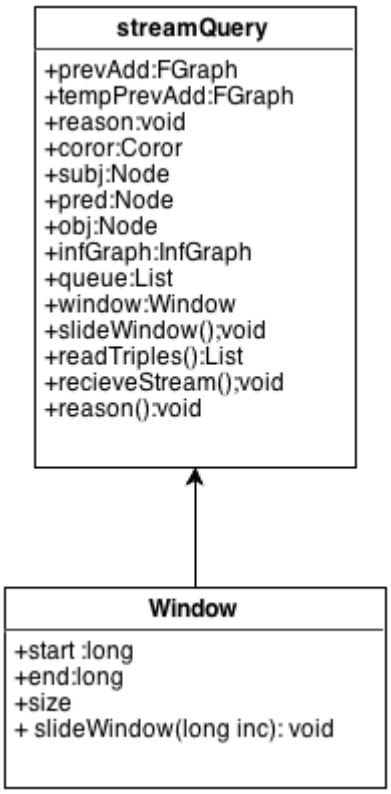


Figure 5-2 Class Diagram for Bare Implementation

Our window class acts as the interval for which triples are valid. Our stream query is then responsible for sliding the window across in 1 second intervals and then retrieving triples from this window(slideWindow() and retrieveTriples()).

Detecting New Triples

When we have a window which we slide across in intervals we come across the problem of having duplicate triples in subsequent windows, as our readTriples() method will return all triples contained in the window at that point, and these would then be inserted into the reasoner. The solution to this is to assign all triples in current window to a graph and then remove any triples that were contained in the previous window (prevAdd). This is where we use the field prevAdd,it is a field kept for storing the previous snapshot.In the implementation the queue can be thought as the stream, all triples received are placed into it, the window is then applied to this queue in order to retrieve the set of current triples. Finally the reason() method is called which involves making calls to the Coror reasoners interface in order to complete the reasoning process. The process runs constantly

making a call to slideWindow() in every iteration in order to move the window along the stream.

Everytime a new window is given to the reasoner, the stream processor will check for all new triples that are located in the window, it will then assign a timestamp to all of these triples. So for example all triples contained in the first window will be given a timestamp of 1 and then inserted into the graph. There are therefore two different timestamps used on the triples, the timestamp the triple is given when it’s in the stream and then also the timestamp it is given when it arrives at the reasoner. The main reasoning behind this method was mainly due to the fact that in the original C-SPARQL implementation is that there is no way of retrieving the windows current position, this methodology then followed into the bare stream reasoning methodology.

5.6 Stream Processing-Coror Integration

Using a sliding window leads us often having overlap between windows, we do not want to re-add these triples to our reasoner. Instead of creating a system to check what triples were contained in previous windows it was decided to use a tumbling window in the stream processor. Specifically a tumbling window with a range of one second. This way triples timestamps can be easily determined by the reasoner and the concept of window is kept in the reasoner. So for instance if a triple arrives at the reasoner at time=2s,then the reasoner (if the window size is 5s say) will assign an expiration time of 7s to that triple.

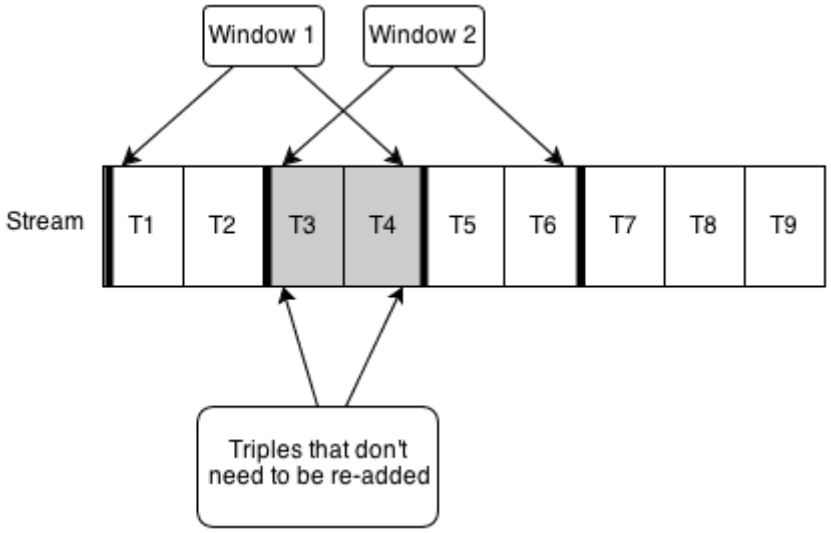


Figure 5-3 Overlap between adjacent Stream Windows

So in the above example, the triples in each compartment T1, T2, T3etc. will be added at separate time, and will all be added to our graph with their respective expiration time.

Once the triples have been added to fadd, the reasoner will then execute any deductions that these triples fire, this process will then repeat until no more new triples are introduced to the any of the reasoners graphs. Once this is done the reasoner will sweep through itsgraphs by iterating through its triples and removing any triples who expiration times are less than the current time, we use the method performDeleteTimeTriples(Temporal Triple, int expirationTime) which removes any copies of the triples from the both raw and the deductions graph whose

timestamps are below the given time. The network will then be swept allowing in order to remove any tokens from the RETE network that have now expired using out sweepRETE(Int i) method, which allows us to iterate through all alpha nodes, followed by all the alpha nodes continuations, if the alpha nodes continuation is a beta node then its queue will be searched and any expired tokens will be removed.

5.7 Stream Coror

Figure 5-4is the Structural Diagram of Coror taken from [12].

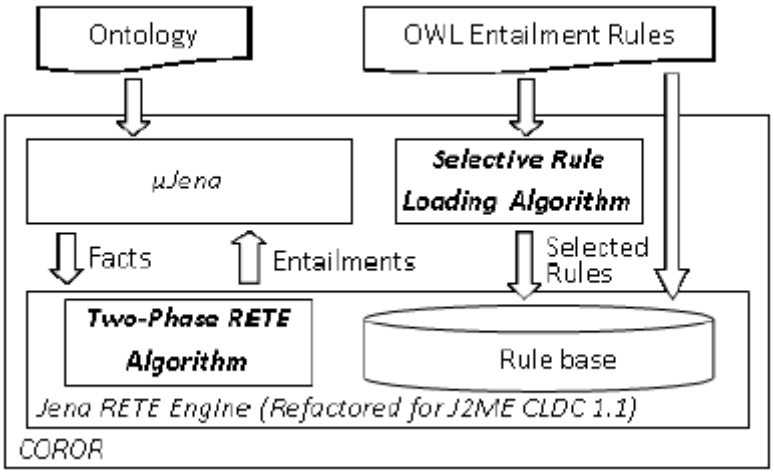


Figure 5-4 Structural Diagram of Coror

Graphs are used to store triples. Allowing us to add and remove triples to them as we please. We can bind a graph to a **Reasoner** if we wish to perform reasoning on the given graph, the reasoner will have a list of rules which it will apply to the graph in order to come to its conclusions.

Temporal classes that needed to be added were Temporal Triples and also Temporal Tokens, tokens being what are stored in the RETE network.

The sequence of events is detailed below in Figure 5-5:

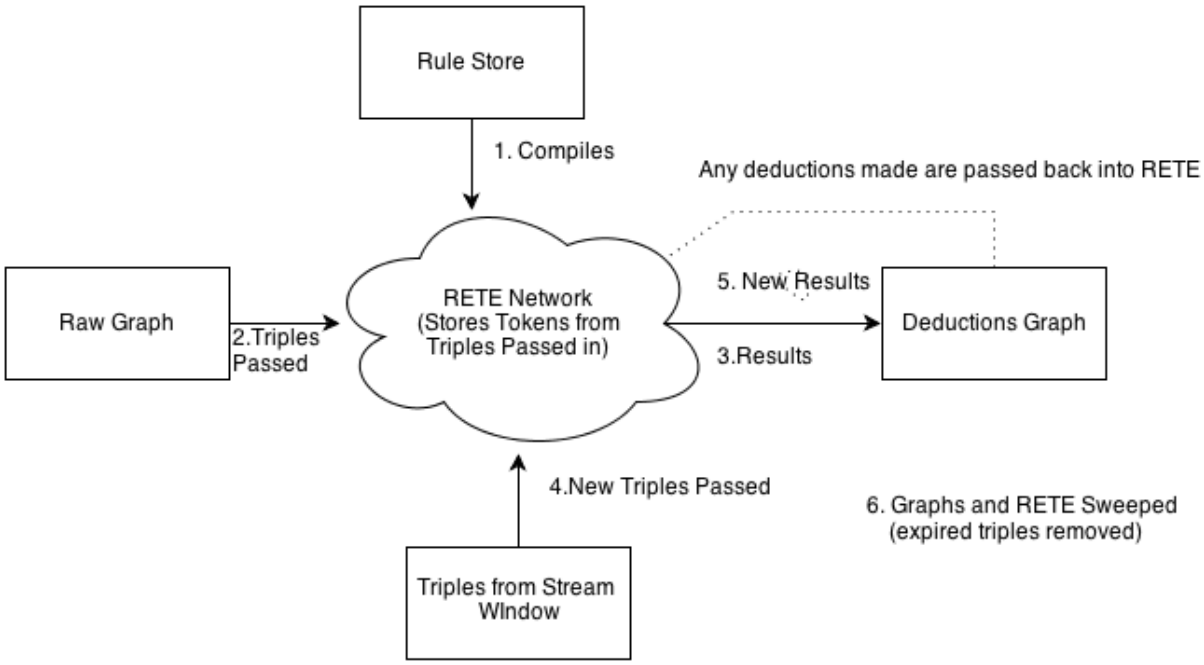


Figure 5-5 Stream Reasoning Process, including Initialization

Coror creates the RETE network by reading through the rules contained in its rule store (.txt file), once the RETE network is created it will through its current base ontology and pass all these triples into the RETE network, the RETE network beta nodes will store any tokens that pass through it and be able to join these facts with any future tokens that pass through them and match the joining criteria, Coror is currently only compatible with the N-Triple syntax for reading triples and ontologies.

For stream Coror needed to firstly be modified in order to make sure it could support incremental reasoning, as the original reasoner would wipe out any deductions made and start the entire reasoning procedure again in the case of new triples being added to the base ontology.

5.7.1 Graph Level

Coror stores triples into graphs which can then be accessed by add, remove or delete operations. There are many different graphs used in Coror. **InfGraph** contains all deductions and raw facts of the reasoner, in other words it contains all the information that the ontology has after reasoning has been performed. There arethen the classes rawGraph(contains all original triples) and **Deductionsgraph** (contains all deducted triples).

Coror contains a Triple class that contains a subject predicate and object. This class was extended to create the object TemporalTriple, which has the additional field Time that indicates when a TemporalTriple will expire.

A new Graph was also required in the modifications of the Coror reasoner, which is a Graph called **fadd**. The purpose of this graph is to hold all new triples that are to be added from the current window. The stream processor is responsible then for adding and removing the necessary triples from/to this graph.

As Coror was originally a reasoner that could only work given a static ontology it did not contain any method for inserting new triples into a pre-built RETE network, any reasoning calls would lead to a complete rebuild of the RETE network and in some cases would cause the RETE network to expand at every iteration. In order to get around this problem the new graph **fadd** was made, this gave a place to store all incoming triples, after the first reasoning call is made causing Coror to build its RETE network any subsequent calls will then pass all the triples contained in **fadd** into the RETE network as opposed to the base ontology.

In the original Coror implementation when a prepare call was made on the inference graph attached to the reasoner, this would cause the entire graph to be passed into the reasoner again,so all the triples that were contained in the inference graph would be fired into the RETE network.

Now when a prepare call is made (after the initial prepare call) instead of passing in the entire inference graph again to the RETE network, it will only pass the graph **fadd**. On the next page is some abbreviated code to give an idea of implementation.

```
public synchronized void prepare() {
    if (isPrepared) return;
    isPrepared = true;
    …
    if(r==0){
    …
       engine.init(true, fdata,true);
    …
    }
    else{
              engine.init(true, fadd,false);
         }
    r++;
  }
```

The new functionalities that had to be implemented into the graph level operations were:

1. Coror must be able to assign timestamps to triples (Temporal Triples)
2. When Coror adds a new triple to the graph all older versions of that triples are removed from the graph.

This was implemented into the base graph class graphImpl, a simple check is done on the graph when a temporal triple is added to any graph to see if there are any triples that must be removed or to see if the triple is already contained in the graph with a greater timestamp. This is implemented in the base class, graphImpl, to ensure that all graphs will perform this functionality.

```
public void performAdd( Triple t ) {
            if(!this.contains(t)) {
            triples.addElement(t);
            . . .
            }
        else if(this.contains(t)&&t instanceof TemporalTriple){
            . . .
           if(tt.getTime()<tta.getTime()){
                triples.remove(t);
                triples.addElement(tta);
             }
          }
```

A similar change had to be made to the RETE terminal nodes, as when a node reached the end of the RETE network it then uses a graph contained in a **RETEConflictSet** in order to determine whether it needs to add a triple to a graph or if the graph already contains, the above code was then re-used. Originally this part of the implementation replaced triples even if they were in not temporal triples. This leads to a large performance loss as replacing non-temporal triples with temporal ones then has a knock on effect with the triples deductions and can make reasoning very slow.

5.7.2 RETE Level

A similar object to TemporalTriple was introduced at the RETE network level, this was the TemporalPBV, this allows timestamps to be associated with a token. A token consists of an array of nodes (subject, predicate or object), the timestamp allows us to remove/update any old tokens.

The RETE network implementation is created using RETEClauseFilterNSs (Alpha nodes), RETEQueueNS (Beta Nodes). ClauseFilters will filter triples that arrive at them and convert them into tokens which they will pass on to their continuations.

It should be pointed out that in the Coror implementation it is not a typical RETE network. There is only one “layer” of filtering, so a triple will only pass through one alpha node instead of passing through a sequence of them. If we are to imagine it then our network would be much wider and shorter than a conventional RETE network.

Each RETEQueueNS object is associated with list, the node has a sibling then this sibling will also be a RETEQueueNS. Both of these RETEQueueNSs contain lists which have all tokens which the node currently holds. The lists are checked with each other and if any joins can be made then they will be executed. In the original version of Coror, whenever a token arrived at a node, this node would check this token with every entry in its sibling list. This can be very inefficient, which is why a tabling method was implemented, we will highlight this approach below.

Firstly an example of joining tokens without tabling:

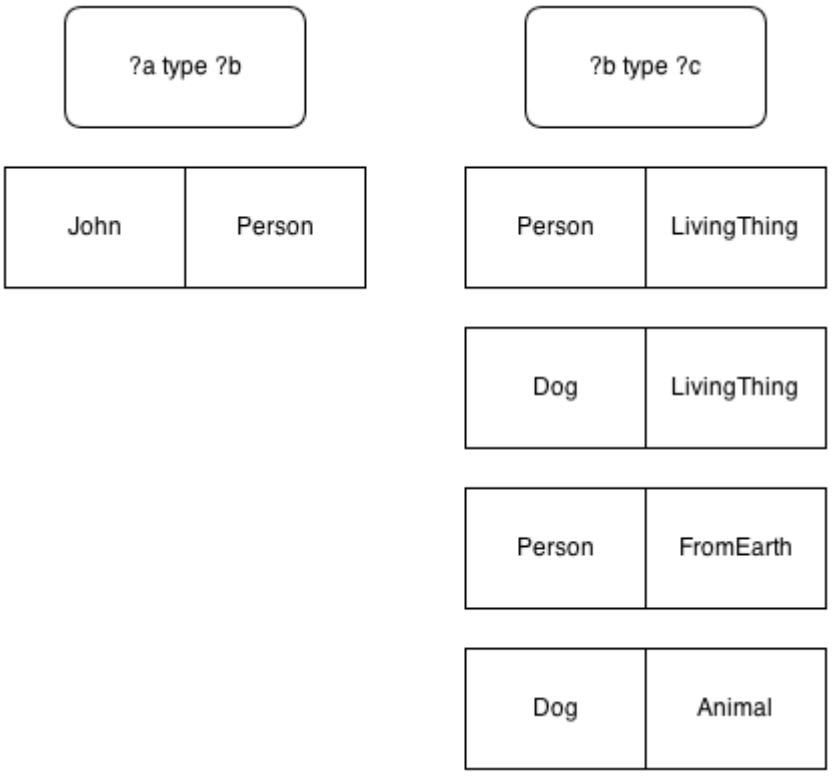


Figure 5-6 RETEQueueNS with Tokens

In Figure 5-6 the token node on the left will search through its sibling on the right in order to find a match (in this case it's looking for a match with its second entry with its siblings first entry. It will cycle down through the tokens and in this case will join tokens to create (John, LivingThing) and (John, FromEarth). The downside of this approach is that in some scenarios it will need to search through many unrelated tokens before it can find any matches. In this case it will always need to search down to the end of the list.

5.7.2.1 Optimized Joins

An optimization made to help with this was done using a table method.

For example all entries that have person in them will be put under the person entry. This way there should be less iterations necessary when completing joins. The above example turns into:

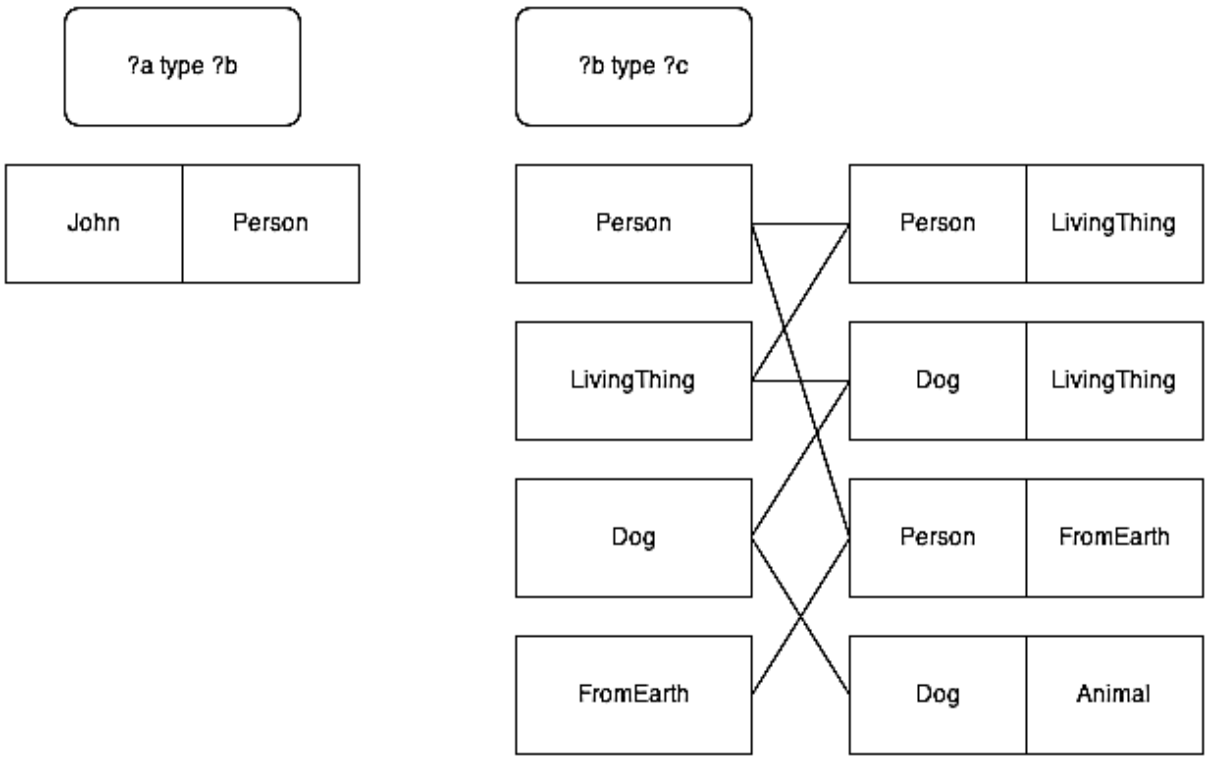


Figure 5-7 Diagram of Optimized Joins

In Figure 5-7 the left node will search through the list on the right and when it finds the relevant entry it will iterate through all the values mapped to it. This means that the implementation does not need to iterate through all the values of the list every time and can lead to a significant decrease in reasoning time depending on the ontology.

These tokens may or may not be temporal tokens (tokens derived from temporal triples). If the tokens are temporal tokens then they will expire and will have to be removed from the network. These is implemented simply by iterating through all alpha nodes and then iterating through all of their continuations, removing any tokens that are now expired. Since a token, say, (dog, animal) will be stored under two different entries, dog and animal, when the token is found to be out of date under one of the mapping entries, it will be removed from the other entries as well.

Similarly we need to update tokens in the case of newer versions of the tokens arriving into the network. This works by running a check on the RETEQueueNS node which receives the token, the node will check all the entries in its list and see if it contains the token, if so and it needs to be updated then it will do so.

In the earlier versions of stream Coror everytime the window of triples was passed to Coror, Coror would assign and update every triples timestamp in that window in its graph. So any triples that did not have the current timestamp were removed from the graph.

This initially seemed like a simple way to implement stream reasoning, but after some thought it was realised that this is in fact just a slower version of the naive approach, as we have to reason every single triple that arrives in our window, and also have the extra overhead that is associated with any sort of temporal reasoning (i.e. the additional operations we describe above).

A previous requirement that was thought to be required(and also common misconception) is that people think that if a fact has multiple derivations and detecting the multiple derivations, as say if one expires but the other derivation still holds then you must be aware that this fact has another derivation and then re-insert the deduction into the graph.

This scenario is actually impossible, since when a deduction has multiple derivations then the derivation with the latest timestamp will be what is inserted into the graph. So for example, in Figure 5-8it is shown what happens when tokens are joined, boxes are labelled with the tokens timestamps while the boxes represent the tokens themselves:

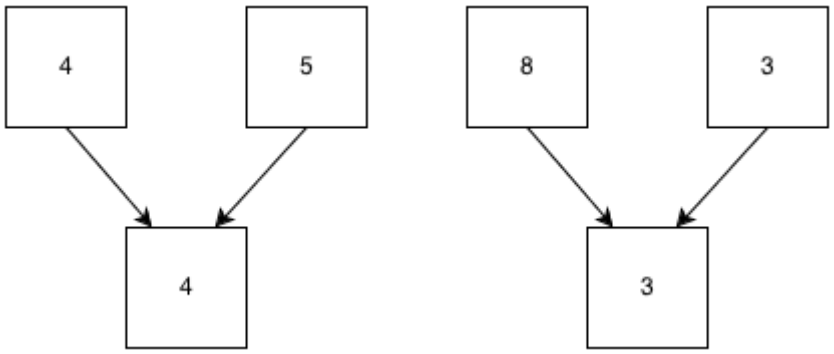


Figure 5-8 Multiple Different Deductions of a consequence

The order of execution is irrelevant, either way the result will be that the resultant triple inserted into the graph will have timestamp of t=4s, since it is higher. As you can see if this deduction were then to expire then also any other deductions must have also expired since by construction the timestamp given to the deduced triple will be the maximum of all of its derivations.

5.7.2.2 Performing Joins

As mentioned earlier, when performing joins between tokens you must be able to entail the result with the minimum timestamp of its two parents, in the case of one of the tokens being a temporal token and the other not, the result will be given the timestamp of the temporal token. This is executed as a simple check on the timestamps in the tokens which are being joined. This functionality is implemented in the following:

PBV newEnv = getNewPartialBindingVector(env, cand, newNodes);

Where env is the token being fired into the node and cand is a token which can be joined with it, newNodes is the set of nodes that will be contained in the resultant token.

5.8 Summary

With the knowledge of how the reasoner has been implemented at the RETE and graphs levels, along with the relevant stream processing and stream generating technique’s, the following section will now look at how the reasoner was evaluated

Chapter 6 Evaluation

In terms of applicability to the kind of domains of interest, the efficiency of the reasoner is crucial, as it must be able to be complete reasoning by the time the next window has arrived. Often comparisons between a stream reasoners and its naive implementation are drawn, in order to show the improvements made by theapproach.The first set of experiments shows the amount of triples added to the ontology from the stream window before the 1second interval is overlapped, this is to give an idea of at what stage the reasoner could no longer reasonably perform stream reasoning for a 1 second window interval.

Inorder to examine how the efficiency of the proposed solution is affected by differing variables such as size of ontology, window size and ontology expressivity, several experiments were devised to measure the speed of the reasoner.The memory consumption of the reasoner is also measured in order to support the statement that this reasoner is capable of being deployed in memory constrained settings.

All experiments were performed on a HP Pavillion DV6, Memory: 4096MB RAM, Processor: AMD Athlon (tm) II P340 Dual-Core Processor (2 CPUs), ~2.2GHZ.

6.1 Experiments

Due to the variable nature (in terms of expressivity and size) of ontologies and in order to examine the effect of window size, several experiments were undertaken to evaluate the solution implemented. All stream speeds unless stated otherwise were set to 1 triple every 400ms.

6.1.1 Experimental Ontologies

A variety of differing ontologies have been used in order to test the reasoner. All of these ontologies were taken from [12] with the exception of the LUBM ontology. Figure 6-1 on the next page gives a table of the ontologies used.

Table of Ontologies

Ontology	Expressiveness	Size (Triples)
LUBM	ALCHI	253
Teams	ALCIN	87
Beer	ALHI(D)	173
mindswapper	ALCHIF(D)	437
mad_cows	ALCHOIN(D)	521
Biopax	ALCHF(D)	633
Food	ALCOF	924
University	SIOF(D)	169
Koala	ALCON(D)	147
isIn	ALCH	6

Figure 6-1 Table of Ontologies Expressivity and Triple Size

[12] used these ontologies as they offer a 1)good variety of domains, 2)They vary in expressivity and 3) They are relatively free of errors and commonly used. The LUBM ontology was also taken as it comes with a data generator that allows for a standard way of producing triples to be streamed.

6.1.2 Throughput Test

If the stream window changes every second we must be able to ensure that the reasoner can complete its reasoning and return results in this time as well.

Three ontologies were tested in order to find the maximum number of additions that could be made to an ontology before the 1 second threshold is breached. The ontologies were the teams, biopax and LUBM ontologies. These were picked as they are a good range of complex to simple ontologies, with teams being the simplest, LUBM being average and biopax being complex.

The results for reasoning time with respect to triples added at that iteration are shown below for the 3 differing ontologies. The stream speed was varied in order to get the required amount of adds to surpass 1 second.

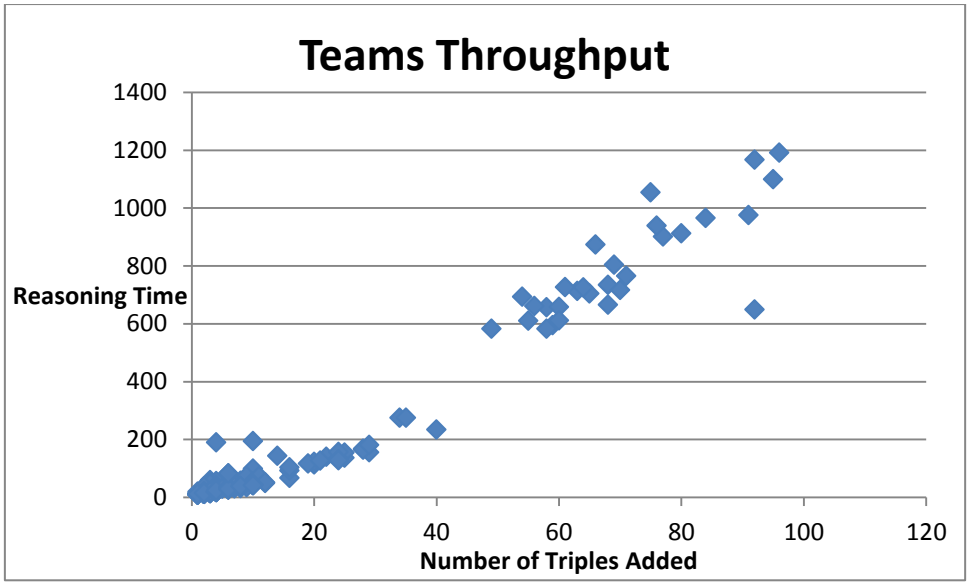


Figure 6-2 Reasoning time vs Number of Triples added for Teams ontology

In the graph above we can see that the teams ontology breaches the 1 second mark at 75 additions.

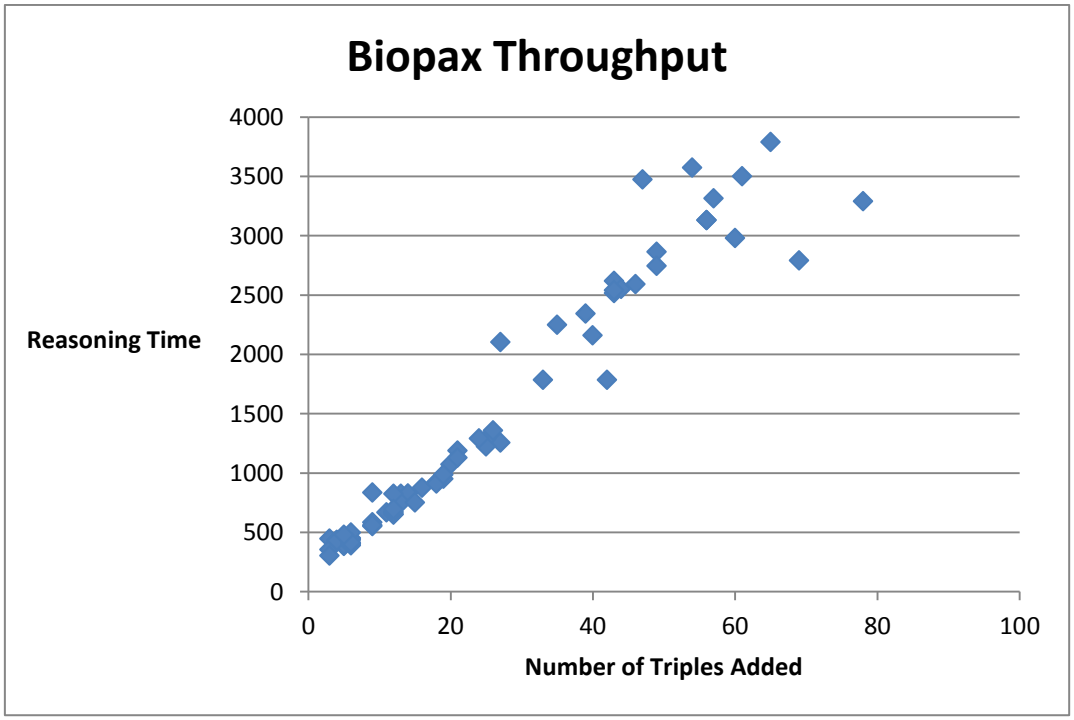


Figure 6-3 Re-Reasoning time vs Number of Triples added for Biopax Ontology

The biopax ontology breaks the 1 second mark first at exactly 20 additions, much less that the teams ontology as expected.

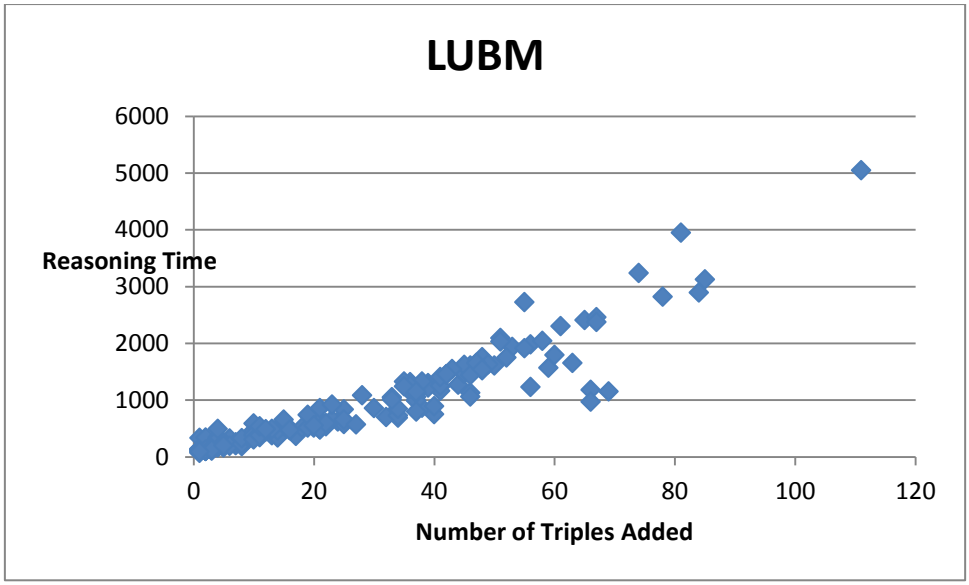


Figure 6-4 Re-Reasoning time vs Number of Triples Added in LUBM ontology

Finally the LUBM ontology above breaks the 1 second mark at 28 additions. The LUBM lands in the middle of the biopax and teams ontologies, also as expected.

From these results can see that the throughput of the reasoner depends on the ontology as well as the window size/speed of stream. Therefore in order to determine the reasoners capability for a particular setting, the peak stream speed should be examined along with the size of the base ontology in order to determine whether that reasoner can adequately reason over the incoming data. Also it can be seen that for all ontologies there is a linear relationship between reasoning time and the number of triples added.

6.1.3 Comparison with Original Implementation

It is difficult to compare the reasoner created with the original reasoner in [5]. However in their evaluation section they compare their reasoner with a naive implementation, a naïve implementation is one where previous deductions are used, and the base ontology and triples in the current window are reasoned over from scratch. It was found that when the ontology consists of more than 13% temporal triples that the naïve approach would out perform their timestamp entailment method. Similarly when tested it was found that with the Dred method, the naive approach would outperform after 2% in [5].

To try and give a flavour of how our reasoner performs in comparison an ontology with a similar reasoning time over the base ontology to the one they tested was chosen. This was the LUBM ontology.

Using the stream reasoner, it was observed if the naive solution would outperform our stream reasoning approach. Results are shown below. The stream speed was varied to give differing amounts of temporal triples contained in the ontology until it reached its maximum frequency.

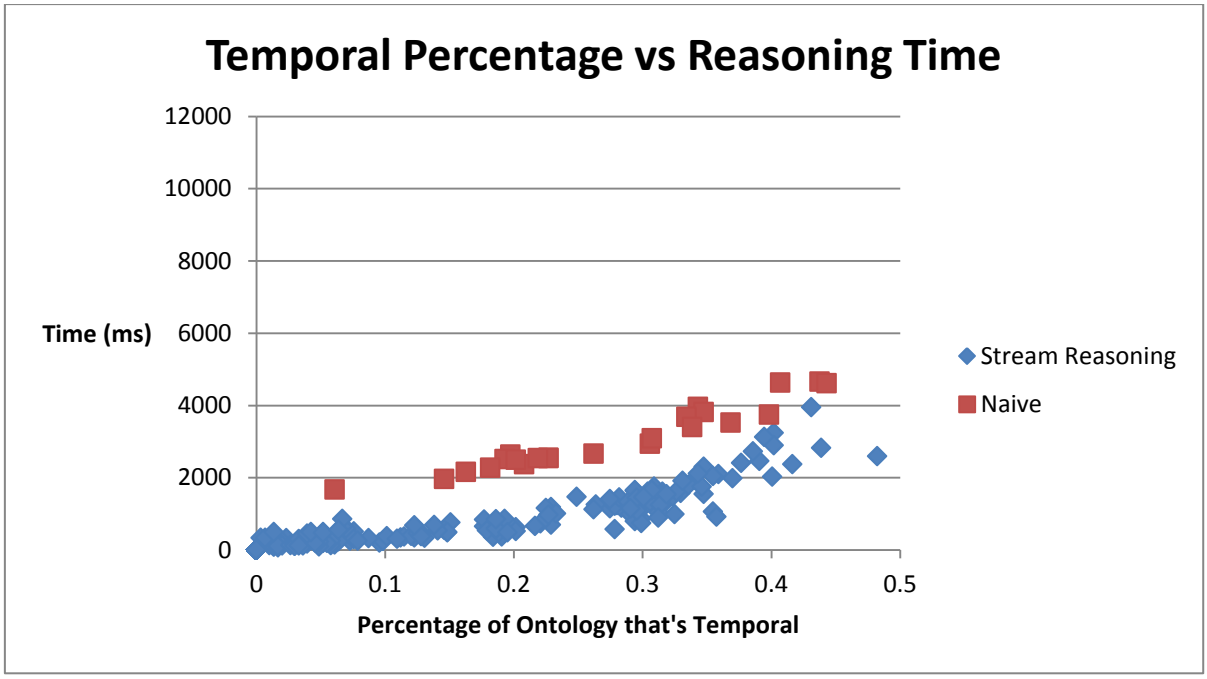


Figure 6-5 re-reasoning time vs percentage of Temporal Triples in Ontology

The results above show reasoning time plotted with respect to percentage of ontology that is temporal. The test was carried out with a 2 second stream window. Up until 40% temporal triples the naive approach never surpasses our stream reasoning approach in terms of reasoning time. This is largely due to the fact that our stream reasoner has much of its timestamp deductions built in to the reasoning process at a minimal cost to speed. The Evaluation could not go higher than the 40% mark as that is the stream at max speed.

6.1.4 Window Variability Experiment

The first experiment was to measure the differing effects of window size on the reasoner. The window size affects how many different temporal relations must be maintained at one time, a large window will generally mean that there will be more temporal triples for the reasoner to process. Also depending on what kind of triples are being passed into the reasoner will cause different types of rules being fired. This means that the type of triple inserted into the stream will affect exactly how much work the reasoner must perform.

The experiment was performed by setting Stream Corors window size to the required time interval and then initiating the reasoner. The time for re-reasoning was recorded along with the amount of changes made due to the triples added at that step.

The ontology used in the following set of experiments was the LUBM ontology,

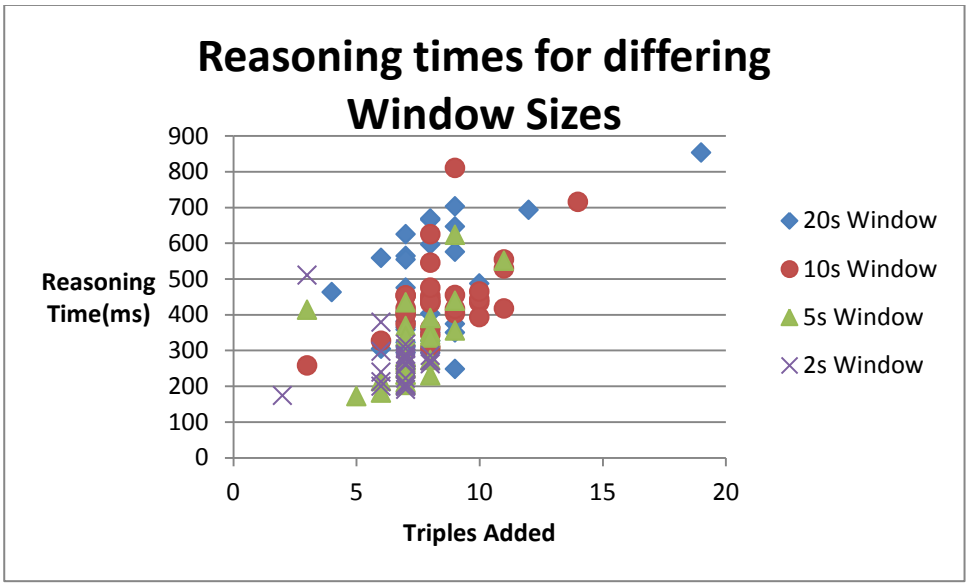


Figure 6-6 Reasoning Time vs Triples added for differing Window Sizes

In the above graph the relationship between reasoning time and the amount of triples added to the ontology for a certain window size are shown.

In Figure 6-6the data is very disperse for all window sizes, this can be attributed to the different types of triples that are being passed into the reasoner and also how many triples the reasoner currently must keep track of (i.e. from previous

reasoning attempts).There is some increase that can be seen in the reasoning time with window size even when there are the same amount of triples being added, this is to be expected as the increase in temporal triples stored in the ontology increases the overall work required by the reasoner. Although we do not see as drastic a difference in the lower window sizes this can be attributed to them being closer in size than the 20 second window, similar amounts of triples were contained in the windows for these experiments. There is also a lot of variability seen between points that have identical number of triples added, this can be largely attributed to the types of facts that are being given the reasoner and what triples reside in the ontology at that point. These factors will lead to differing amounts of subsequent adds to the ontology.

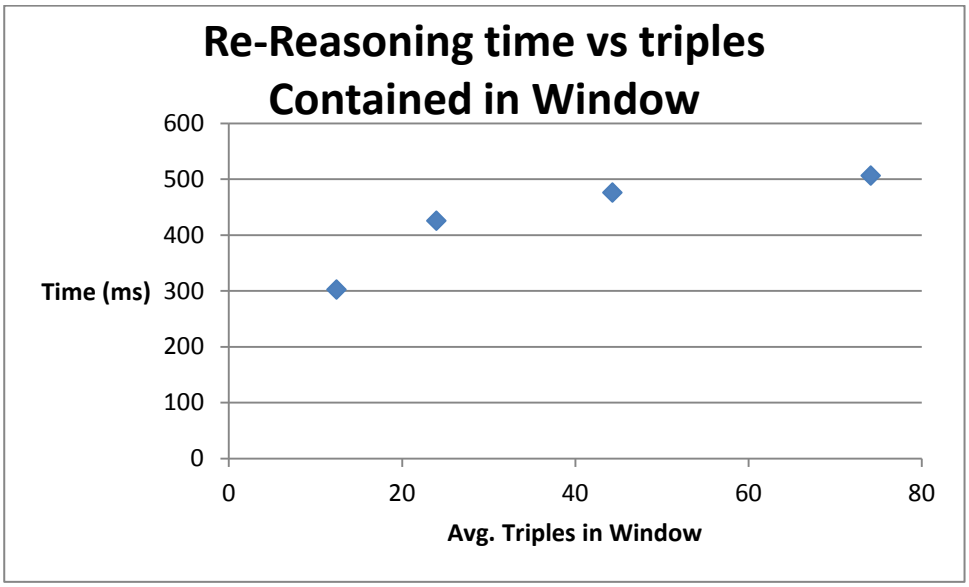


Figure 6-7 Averaged Reasoning Time vs Average number of triples in window, data from Fig 6-6

In the above graph we plotted the avg amount of triples in window and the average reasoning time for a window, these values were averaged from the data in Figure 6-6.

There is a noticeable increase of reasoning time with the average number of triples, there is also an observed fall off as if the curve is levelling off. Since the experiments were averaged over 13 runs, it can take the first few runs to even fill the window fully, in the case of the larger windows they may only have a full

window for the final 3 runs. This is the reason we see the steady decline in slope of the graph.

6.1.5 Differing Ontology Experiments

For stream reasoning the data inside the stream is most certainly not the only variable that we are concerned with, the expressivity and size of the ontology is very important in determining how much work a reasoner must do in processing an ontology. For instance transitive relationships are generally considered to be one of the more computationally taxing properties due to the fact that they can cause changes to propagate widely through an ontology. In this experiment we looked at how Stream Coror performs when given several ontologies with different expressivity. The window size taken in this case was a 10 second time interval, the average of the re-reasoning time was taken over 13 iterations.

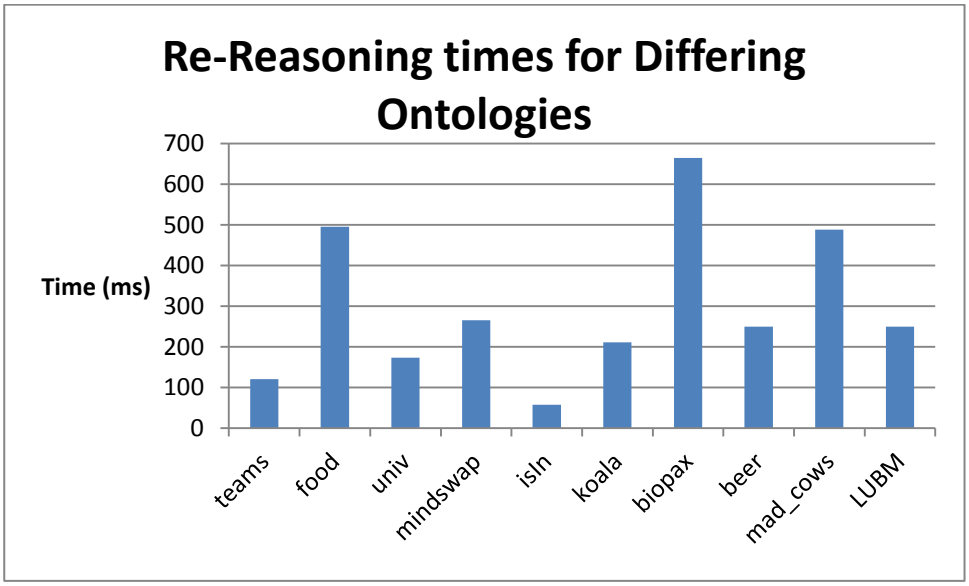


Figure 6-8 Re-Reasoning times for different ontologies

The size and expressivities of the ontologies used aregiven inFigure 6-1.

Looking at 6-7 we can see that the overall size of an ontology has a high effect on the amount of time taken in re-reasoning. This is can be accredited to the amount of searches required to be done when changes are made to the ontology (we need to search ontology for old temporal triples that might need to be updated),

and also for the fact that an high number of triples in the ontology will mean a high number of tokens in the RETE network, so similarly more searches will need to be done on the queues in order to perform the necessary joins.

In order to see how much size contributes to the reasoning time even when using different ontologies, we have included the graph below taken from the same experimental data as above.

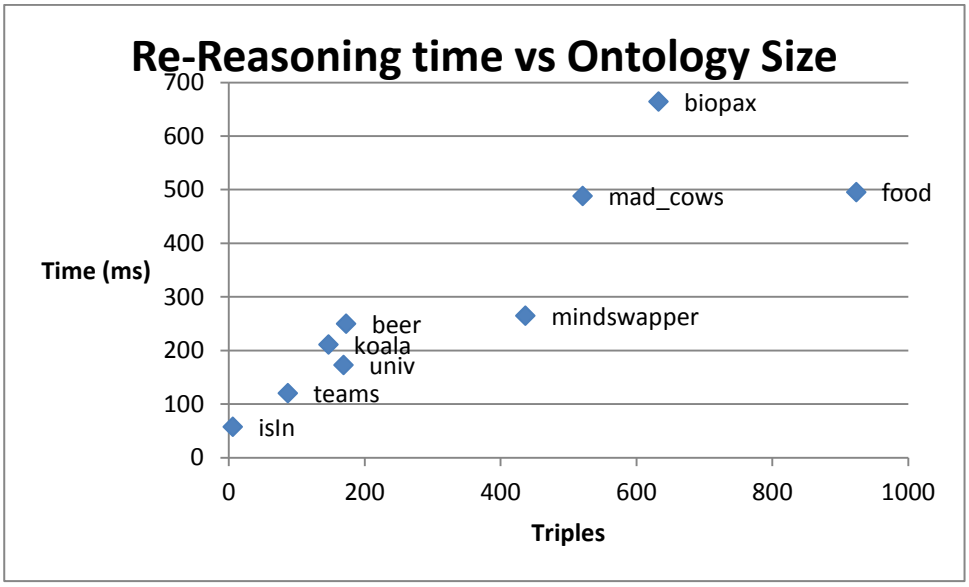


Figure 6-9 Re-Reasoning time vs Triples Contained in Ontology

Above a graph of re-reasoning time vs triples contained in the ontology for all the different ontologies in Figure 6-1.

There is still a noticeable increasing linear trend of reasoning time vs triples contained in the ontology, although it is much more dispersed due at the end since for large triple sizes you could have different types of triples e.g. food ontology only has 10 properties while biopax has 50.Expressivity can be seen to not play the major role in determing reasoning time however, as there is most certainly a more obvious increase with ontology size.

6.1.6 Ontology Size

To see the exact effect of the ontology size with respect to reasoning time we took the mad_cows ontology and varied its size to see how drastic a change occurs. The types of triples introduced were instances of classes defined in the mad_cows ontology.

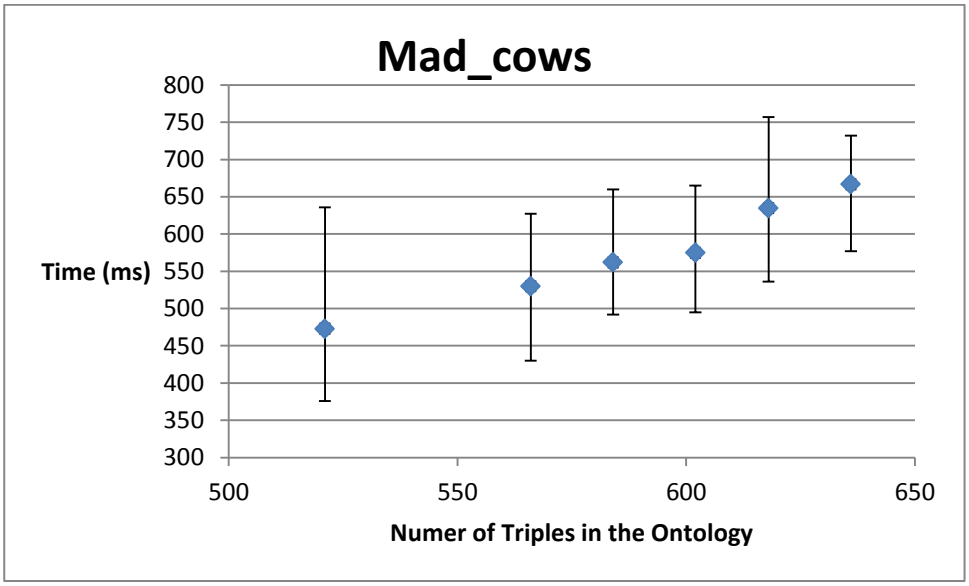


Figure 6-10 Average re-reasoning Time vs Ontology Size (mad_cows Ontology)

Above is a graph for average re-reasoning time vs number of triples in the mad_cows ontology, the triples added to the ontology in this case were instances of classes already defined in the ontology. The time recorded is the average re-reasoning time over 12 iterations and the experiments were performed with a 10 second time window.

There is an increasing linear trend to be seen in triples added. The increase is quite dramatic, while linear the increase is still considerable and can be considered an important factor for Stream Coror.

We also became interested with the idea of varying an ontologies Tbox axioms, as this could more directly affect the triples that are being streamed into our reasoner. This is due to the fact that TBox axioms affect ABox axioms, but ABox axioms do not affect TBox axioms. TBox axioms could then potentially affect more of the

ontology when altered. We used the LUBM ontology and repeatedly ran the reasoner over the steam while removing different axioms on each run. Results are shown below.

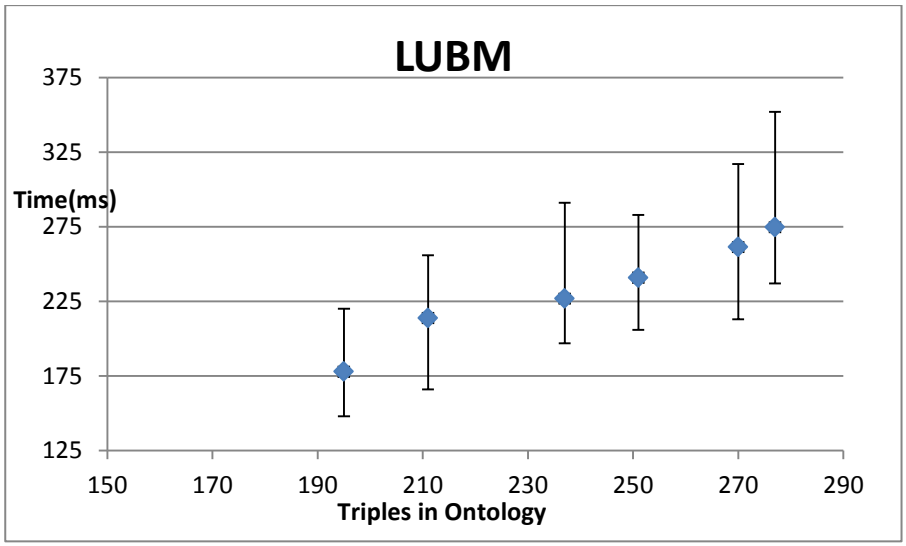


Figure 6-11 Re-Reasoning time vs Ontology Size (LUBM Ontology)

Figure 6-11 shows the average re-reasoning time vs triples in the LUBM ontology. In this case the triples that are removed are TBox axioms. The time recorded is the average re-reasoning time over 14 iterations.

A similar increase can be seen in the above graph, the more TBox axioms means that more deductions are made on the incoming stream leading to an overall increase in reasoning time.

Both graphs exhibits a linear increase with reasoning time and triples in the ontology, while the graph is steeper for adding instances we cannot make any concrete assertions from this as the experiments are over differing ontologies. Even if they were over the same ontology, results would still vary depending on the types of axioms and instances contained in the ontology and which ones are being removed.

6.1.7 Memory

The memory of Stream coror was measured also. The two differing implementations of the stream processing process differ drastically in memory consumption. As C-SPARQl has a very complex query language it takes up a significant amount of space on our reasoner.

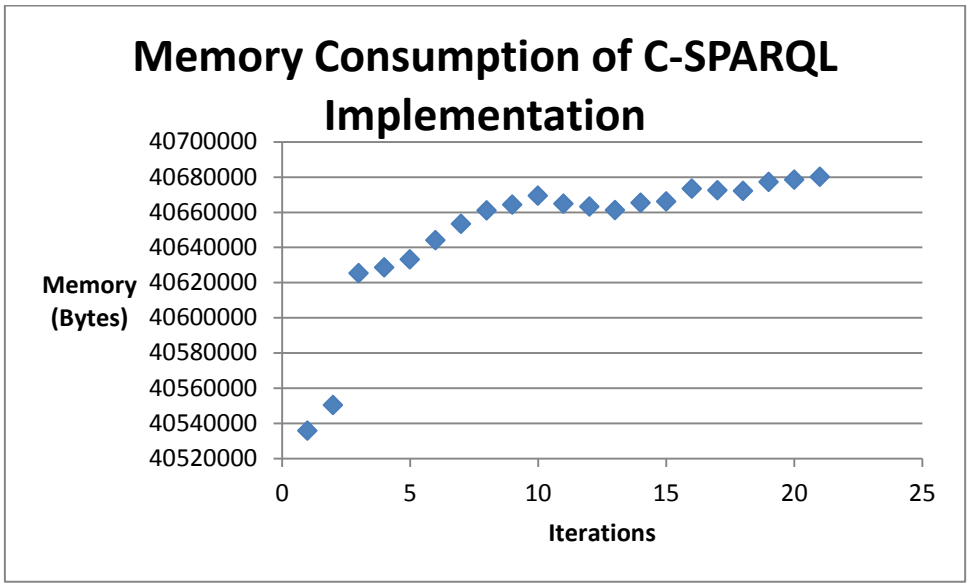


Figure 6-12 Memory Consumption of C-SPARQL implementation

Figure 6-12 shows the memory consumption of our reasoner when using C-SPARQL for the stream processing component. Runs is the number of iterations the reasoner is at (i.e. how many times it has re-reasoned). With additional code optimizations and improved data structures the memory consumption could be reduced further.

Above is the graph of the memory consumption by our C-SPRAQL implementation. This shows a steady increase in memory consumption by the reasoner, as well as having an initially very high memory cost.

In the future of resource constrained reasoning we may encounter problems of having very rich data streams but no efficient way of extracting this information effectively in terms of memory usage.

Below is the memory usage for our bare implementation, as you can see there is significantly less initial memory consumption, this is due to the fact that the bare implementation is very simplistic and merely gives us a way of retrieving triples from a window. It can also be seen that there is much less memory leaking by the implementation. This is invaluable for stream reasoning as in most scenarios we would wish for the reasoner to run continuously, so it is important that performance will not be decreased as time goes on.

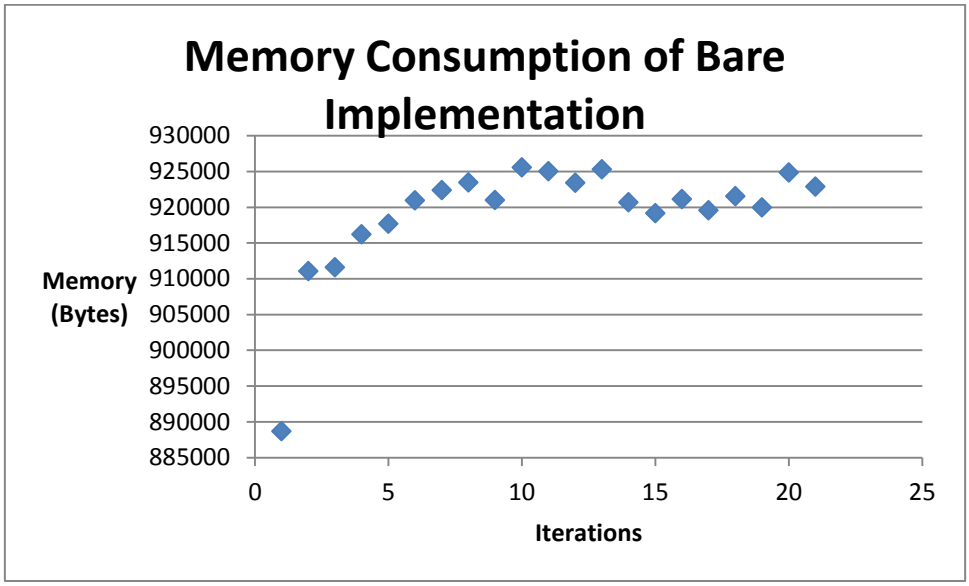


Figure 6-13 Memory Consumption of Bare Implementation

Above is the graph for memory consumption for our reasoner when using our own stream processor. . Runs is the number of iterations the reasoner is at (i.e. how many times it has re-reasoned).

The above graph shows that the bare implementation has significantly less memory consumption than the C-SPARQL implementation, which should be so due to the decreased functionality. It can also be seen that there is less memory leakage over time for the bare implementation, this is very important for stream reasoning.

Note

While the bare implementation may show less memory leaking over time than the C-SPARQL version it should be made clear that for a stream that comes in

extremely quickly both implementations will run into problems, merely because the stream processor will hold all information it hasn't processed yet in memory, if the stream comes in quickly this can cause the reasoner to buffer extremely large amounts of information and cause memory consumption to increase over time.

6.1.8 Overall Summary of Findings from Experiments

In the evaluation it has been found that the stream reasoning approach provides significant performance increases over its naive counterpart, and also performing favourably when compared to the original implementation in [5].

The throughput experiments showed significant differences between breaching the 1 second mark for differing ontologies, which lead us to more experiments in order to try and gain understanding of factors that can lead to increased re-reasoning time. Size of the base ontology was found to be a significant factor while expressivity played a much more minor role as was seen in section 6.1.4. This is outlined further in section 6.1.5, showing significant increases in reasoning time with respect to ontology size.

Since differing window sizes are required, dependant on the scenario the reasoner is in. Section 6.1.3 looks at this variability, seeing that the window size can lead to a noticeable increase in reasoning time. This again leads us to the result that the throughput of the reasoner would need to be reduced in order to stop the reasoner from being overburdened.

The fact that there are such strong relations between reasoning time and the above mentioned variables, this means the throughput will also be affected. Due to the significant trends that can be seen between reasoning time and these variables, it can be said that the throughput for a reasoner can more safely determined once the reasoner is tested with the desired ontology, once the relevant trends are found they can be used to decide on how to attain a suitable throughput.

Chapter 7 Conclusion

In the current SOA there a few possible approaches to stream reasoning, some have pursued an approach adopting truth maintenance systems, where as a timestamp entailment approach has been taken in this paper similar to [5]. The timestamp entailment approach had not been applied to a variety of rules before now, nor has it been used over well-known ontologies. This paper has given a simple method for altering RETE based reasoners to support stream reasoning without the need for external maintenance system. This is, as far as this work is concerned, is the first work that supports the timestamp entailment approach at a reasoning level, currently implemented in a resource-limited reasoner.

This method for stream reasoning has proven to be a viable method for future reasoners to be based upon, asall results have shown competitive results with other stream reasoners with the reasoner constantly outperforming its naive counterpart at all times up to 40%, as opposed to other reasoners becoming outperformed by their naive counterpart at 13% and 0-2%.

In terms of being able to perform stream reasoning effectively, it has been shown at what stage the throughput of our reasoner was no longer able to perform sufficiently in section 6.1.2. The throughput was found to be related with window size and the base ontology being reasoned over, this helps to give an idea of the reasoners limitations while also leading to further tests in order to examine what exact relationships of an ontology lead to a differing result in throughput and if any of the variables can are considerably more sensitive to change than others.

Various variables were altered in the experiments in order to see how the reasoner would react, trends were seen for triples added to the ontology from the stream window, window size and also ontology size. Well defined linear trends were drawn from each of these variables, these trends can help to predict the reasoners capabilities, and to help others in order to compare their implementations to this reasoner. Variability of data in experiments was noticed, this is accredited to the fact that differing triples can be sent to a reasoner, along with differing amounts of triples in the window as well. It has also been seen that with these linear trends, it is believed that they are both of significant importance

when considering how effective the reasoner will be in a certain scenario, and one cannot be put in front of the other as a main cause of increasing reasoning time.

Ontology expressiveness did not play as significant a role as ontology size did in the evaluation, this can be accredited partly to Corors semi-limited rule set but also to the RETE network, since as the fact base grows more joins need to be checked in the network increasing work as the ontology grows.

The fact that temporal relationships between triples can be maintained with modifications to the reasoning algorithm leads us to no longer require large external maintenance systems to deduce complex temporal entailments, this technique does not need to limit itself to resource limited environments as it can be applied to any reasoner that uses a RETE network.

7.1 Future Works

7.1.1 Indexing

Currently in order to remove expired triples and tokens from reasoner coror must search through every entry in the given graph/node. Indexing could help Coror substantially in helping sweep out expired tokens and triples from the graphs and RETE network. This can help in decreasing overall reasoning time. Similarly in the joins performed between tokens in the RETE network, currently our tabling method works by creating a link between entries in a token, with the token itself. The problem is even entries which are never joined are put into this table, this means our implementation must read through the queue an unnecessary amount of times. If we could implement an approach where only entries that are used are placed in the map this could improve reasoning performance significantly.

7.1.2 Hybrid Reasoning

Coror is a pure forward reasoner, many reasoners use a hybrid approach that uses both forward and backward reasoning, in the future if a suitable backward approach is created in order to entail timestamps it may be composed with our reasoner in order to increase performance. This could also help reduce memory consumption, as a backward reasoner to run less materializations need to be kept in memory at one time, this could be a very viable setting for some resource limited stream reasoner.Implementing a hybrid approach would reduce the number of alpha nodes required in the RETE network, so less checks would need to be done overall on triples entering the network.

7.1.3 Resource Limited Stream Processor

In the evaluation section it was found that the C-SPARQL library used up a considerable amount of memory with respect to our stream reasoner. This highlights the need for a more lightweight way of querying streams. Ideally a configurable stream processor that can allow for certain functionalities to be disabled to allow for simpler or more complex stream processing, depending on what is required.

7.1.4 Full Reasoner

Creating a full OWL stream reasoner could be another potential use for the technique outlines in this work. This could show how this approach deals with more complex ontologies where the full OWL-DL specification can be processed.

7.1.5 Partial Results

As discussed in the State of the Art (5.3), works for returning partial reasoning results in cases where the memory limits is exceeded could allow reasoners to perform more reliably in environments where the rate of information flowing into the reasoner has very high variance, potentially causing out of memory errors. A similar approach could be taken for our reasoner, perhaps removing certain rules from firing if memory consumption gets too high.

Bibliography

[1] W3C OWL Overview, URL: http://www.w3.org/2001/sw/wiki/OWL Retrieved August 2013

[2] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, Dieter Fensel, It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Xplore Digital Library,*December 2009.

[3] Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and

Michael Grossniklaus. Stream Reasoning: Where We Got So Far, URL: http://wasp.cs.vu.nl/larkc/nefors10/paper/nefors10_paper_0.pdf Retrieved August 2013

[4] Gulay Unel and Dumitru Roman. Stream Reasoning: A Survey and Further Research Directions. *Lecture Notes in Computer Science Volume 5822, 2009,* pp 653-662.

[5] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus Incremental Reasoning on Streams and Rich Background Knowledge. URL: http://www.larkc.eu/wp-content/uploads/2008/01/2010-Incremental-Reasoning-on-Streams-and-Rich-Background-Knowledge.pdf, Retrieved August 2013.

[6] Kia Teymourian, Malte Rohde, Ahmad Hasan, and Adrian Paschke. Fusion of Event Stream and Background Knowledge for Semantic-Enabled Complex Event Processing. Challenge Paper. URL: http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Workshops/DeRiVE/derive2011_submission_4.pdf Retrived August 2013.

[7] Charles L. Forgy.Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. URL:http://www.csl.sri.com/users/mwfong/Technical/RETE%20Match%20Algorithm%20-%20Forgy%20OCR.pdf, Retrieved August 2013.

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

[9] Darko Anicic, Sebastian Rudolph, Paul Fodor, Nenad Stojanovic. Stream Reasoning and Complex Event Processing in ETALIS.*Semantic Web 1 (2009) 1–5 1IOS Press*

[10] Luke Steller, Shonali Krishnaswamy, Simon Cuce, Jan Newmarch and Seng Loke. A WEIGHTED APPROACH FOR OPTIMISED REASONING FORPERVASIVE SERVICE DISCOVERY USING SEMANTICS AND CONTEXT.*ICEIS (4), 2008 - researchgate.net*

[11] Steller, L., Krishnaswamy, S. and Gaber, M.Enabling scalable semantic reasoning for mobile services. *2009 - eprints.port.ac.uk*

[12] W Tai, J Keeney, D O'Sullivan. Resource Constrained Reasoning Using a Reasoner Composition Approach. se*mantic-web-journal.net. 06/10/2013*

[13] Y Ren and JZ Pan. Optimising Ontology Stream Reasoning with Truth Maintenance System.*Proceedings of the 20th ACM international conference, 2011.*

[14] H Stuckenschmidt and S Ceri. Towards Expressive Stream Reasoning. *Dagstuhl Seminar Proceedings 10042. 2010.*

[15] The University of Texas at Austin. URL: http://homepage.psy.utexas.edu/homepage/Faculty/Markman/PSY305/PDF/Reasoning1.pdf. Retrieved August 2013

[16] W3C RDF Overview. http://www.w3.org/RDF/. Retrieved August 2013

[17] RDF Vocabulary Description Language 1.0: RDF Schema URL: http://www.w3.org/TR/rdf-schema/. Retrieved August 2013

[18] Yuan Ren, Jeff Z. Pan, and Yuting Zhao.Ontological Stream Reasoning via Syntactic ApproximationURL: http://people.csail.mit.edu/pcm/tempISWC/workshops/IWOD2010/paper3.pdf. 2010

[19] OWL Web Ontology Language Guide URL: http://www.w3.org/TR/owl-guide/. Retrieved August 2013.

[20] W3C RDF Semantics. URL:http://www.w3.org/TR/rdf-mt/ . Retrieved August 2013.

[21] Ralf Moller and Volker Haarslev. http://www.sts.tuharburg.de/papers/2009/MoHa09.pdf . *Handbook on Ontologies, 2009*.

[22] B Suntisrivaraporn. Module Extraction and Incremental Classification: A Pragmatic Approach for EL+ Ontologies*.The Semantic Web: Research and Applications, 2008.*

[23] Darko Anicic, Paul Fodor, Sebastian Rudolph and Nenad Stojanovic. URL:http://www.aifb.kit.edu/images/c/c0/Www29-anicic.pdf. *Proceedings of the 20th international conference on World wide web Pages 635-644.*

[24] Jie Bao, Caregea D and Honavar V. A Tableau-based Federated Reasoning Algorithm for Modular Ontologies. *Web Intelligence, 2006. WI 2006. IEEE/WIC/ACM International Conference. 2006.*

[25] National and Kapodistrian University of Athens. URL: http://cgi.di.uoa.gr/~pms509/lectures/tableaux-techniques2spp.pdf. Retrieved August 2013,

[26] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri and Michael Grossniklaus. An Execution Environment for C-SPARQL Queries. *Proceedings of the 13th International Conference on Extending Database TechnologyPages 441-452*

[27] Freddy Lecue, Spyros Kotoulas, and Pol Mac Aonghusa. Capturing the Pulse of Cities: A Robust Stream Data Reasoning Approach*. IBM Research Smarter Cities Technology Centre.2011.*

[28] Emanuele Della Valle, Davide F. Barbieri,Daniele M. Braga, Stefano Ceri Florian Fischer and Volker Tresp. LarKCFP7 – 215535. URL: http://www.larkc.eu/wp-content/uploads/2008/01/larkc_d33-description-of-

strategy-and-design-for-data-stream-management-approaches_final.pdf. Retrieved August 2013.

[29] W3C OWL Web Ontology Language. URL: http://www.w3.org/TR/owl-ref/. Retrieved August 2013.

[30] J Urbani, S Kotoulas and E Oren. Scalable Distributed Reasoning using MapReduce. *The Semantic Web-ISWC, pp 634-649. 2009*.

[31] T Di Noia, R Mirizzi, VC Ostuni, D Romito. Linked Open Data to support Content-based Recommender Systems. Proceedings of the 8th International Conference on Semantic Systems Pages 1-8. 2012.

[32] M Sabou, AMP Braşoveanu. Supporting Tourism Decision Making with Linked Data. *Proceedings of the 8th International Conference on Semantic Systems, Pages 201-204. 2012.*

[33] A Hogan, A Harth and A Polleres. Scalable Authoritative OWL Reasoning on a Billion Triples.*7th International Semantic Web Conference. 2008*

[34] Payam Barnaghi, Stefan Meissner, Mirko Presser, and Klaus Moessner. Sense and Sens'ability: Semantic Data Modelling for Sensor Networks. *ICT-MobileSummit 2009 Conference Proceedings. 2009.*

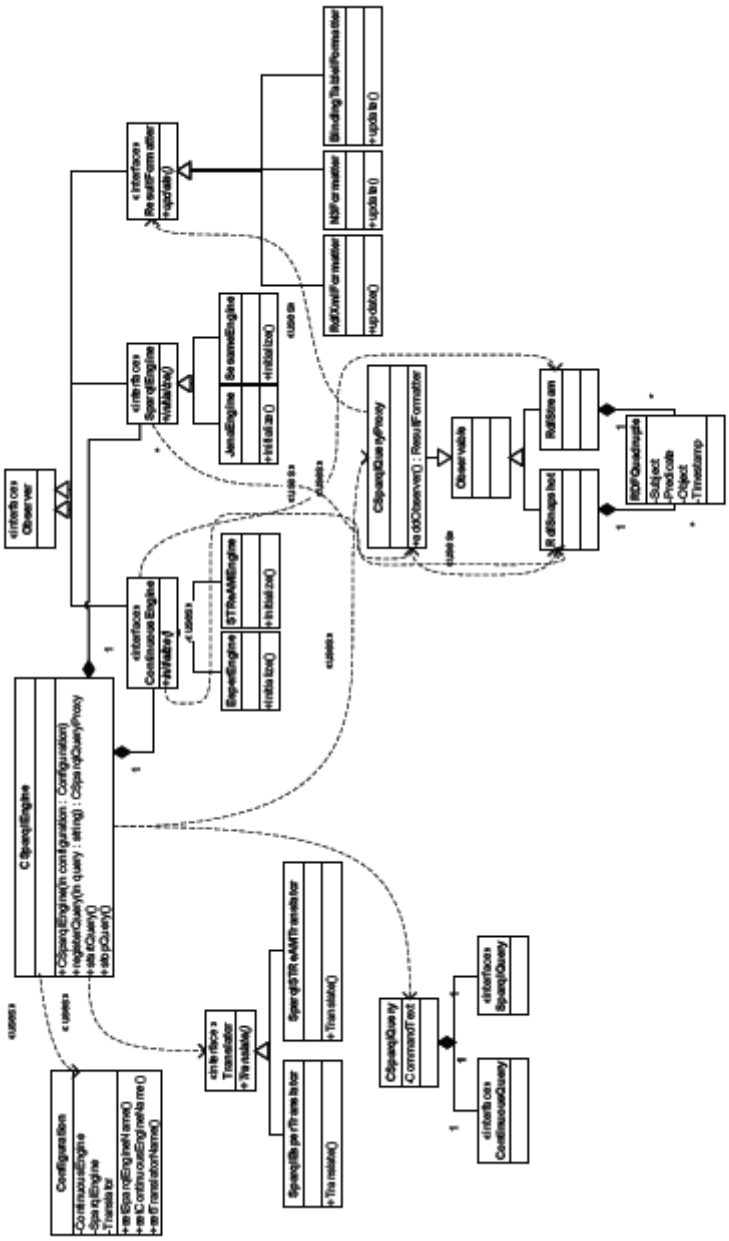
[35] Apache Jena. URL: http://jena.apache.org/. Retrieved August 2013.

[36] Sebastian Rudolph. Foundations of Description Logics. URL: http://www.aifb.kit.edu/images/1/19/DL-Intro.pdf. Retrieved August 2013.

[37] SWAT. The Lehigh University BenchmarkURL: http://swat.cse.lehigh.edu/projects/lubm/. Retrieved: August 2013.

[38] W3C SPARQL query language. URL: http://www.w3.org/TR/rdf-sparql-query/. Retrieved August 2013.

Appendix C



Appendix B

P-entailment rules

[rdfp1: (?p rdf:type owl:FunctionalProperty), (?u ?p ?v), (?u ?p ?w), notLiteral(?v) -> (?v owl:sameAs ?w)]

[rdfp2: (?p rdf:type owl:InverseFunctionalProperty), (?u ?p ?w), (?v ?p ?w) -> (?u owl:sameAs ?v)]

[rdfp3: (?p rdf:type owl:SymmetricProperty), (?v ?p ?w), notLiteral(w) -> (?w ?p ?v)]

[rdfp4: (?p rdf:type owl:TransitiveProperty), (?u ?p ?v), (?v ?p ?w) -> (?u ?p ?w)]

[rdfp5a: (?v ?p ?w) -> (?v owl:sameAs ?v)]

[rdfp5b: (?v ?p ?w), notLiteral(?w) -> (?w owl:sameAs ?w)]

[rdfp6: (?v owl:sameAs ?w), notLiteral(?w) -> (?w owl:sameAs ?v)]

[rdfp7: (?u owl:sameAs ?v), (?v owl:sameAs ?w) -> (?u owl:sameAs ?w)]

[rdfp8ax: (?p owl:inverseOf ?q), (?v ?p ?w), notLiteral(?w) -> (?w ?q ?v)]

[rdfp8bx: (?p owl:inverseOf ?q), (?v ?q ?w), notLiteral(?w) -> (?w ?p ?v)]

[rdfp9: (?v rdf:type owl:Class), (?v owl:sameAs ?w) -> (?v rdfs:subClassOf ?w)]

[rdfp10: (?p rdf:type rdf:Property), (?p owl:sameAs ?q) -> (?p rdfs:subPropertyOf ?q)]

[rdfp11: (?u ?p ?v), (?u owl:sameAs ?up), (?v owl:sameAs ?vp), notLiteral(?up) -> (?up ?p ?vp)]

[rdfp12a: (?v owl:equivalentClass ?w) -> (?v rdfs:subClassOf ?w)]

[rdfp12b: (?v owl:equivalentClass ?w), notLiteral(?w) -> (?w rdfs:subClassOf ?v)]

[rdfp12c: (?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?v) -> (?v owl:equivalentClass ?w)]

[rdfp13a: (?v owl:equivalentProperty ?w) -> (?v rdfs:subPropertyOf ?w)]

[rdfp13b: (?v owl:equivalentProperty ?w), notLiteral(?w) -> (?w rdfs:subPropertyOf ?v)]

[rdfp13c: (?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?v) -> (?v owl:equivalentProperty ?w)]

[rdfp14a: (?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u ?p ?w) -> (?u rdf:type ?v)]

[rdfp14bx: (?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u rdf:type ?v), notLiteral(?p) -> (?u ?p ?w)]

[rdfp15: (?v owl:someValuesFrom ?w), (?v owl:onProperty ?p), (?u ?p ?x), (?x rdf:type ?w) -> (?u rdf:type ?v)]

[rdfp16: (?v owl:allValuesFrom ?w), (?v owl:onProperty ?p), (?u rdf:type ?v), (?u ?p ?x), notLiteral(?x) -> (?x rdf:type ?w)]

D* entailment rules

[lg-rdfs1: (?v ?p ?l), isPLiteral(?l), assignAnon(?l, ?b) -> (?v ?p ?b), (?b rdf:type rdfs:Literal)]

[lg-rdfs2D: (?v ?p ?l), isDLiteral(?l, ?t), assignAnon(?l, ?b) -> (?v ?p ?b), (?b rdf:type ?t)]

[rdf1: (?v ?p ?w) -> (?p rdf:type rdf:Property)]

[rdfs2: (?p rdfs:domain ?u), (?v ?p ?w) -> (?v rdf:type ?u)]

[rdfs3: (?p rdfs:range ?u), (?v ?p ?w), notLiteral(?w) -> (?w rdf:type ?u)]

[rdfs4a: (?v ?p ?w) -> (?v rdf:type rdfs:Resource)]

[rdfs4b: (?v ?p ?w), notLiteral(?w) -> (?w rdf:type rdfs:Resource)]

[rdfs5: (?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?u) -> (?v rdfs:subPropertyOf ?u)]

[rdfs6: (?v rdf:type rdf:Property) -> (?v rdfs:subPropertyOf ?v)]

[rdfs7x: (?p rdfs:subPropertyOf ?q), (?v ?p ?w) -> (?v ?q ?w)]

[rdfs8: (?v rdf:type owl:Class) -> (?v rdfs:subClassOf rdfs:Resource)]

[rdfs9: (?v rdfs:subClassOf ?w), (?u rdf:type ?v) -> (?u rdf:type ?w)]

[rdfs10: (?v rdf:type owl:Class) -> (?v rdfs:subClassOf ?v)]

[rdfs11: (?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?u) -> (?v rdfs:subClassOf ?u)]

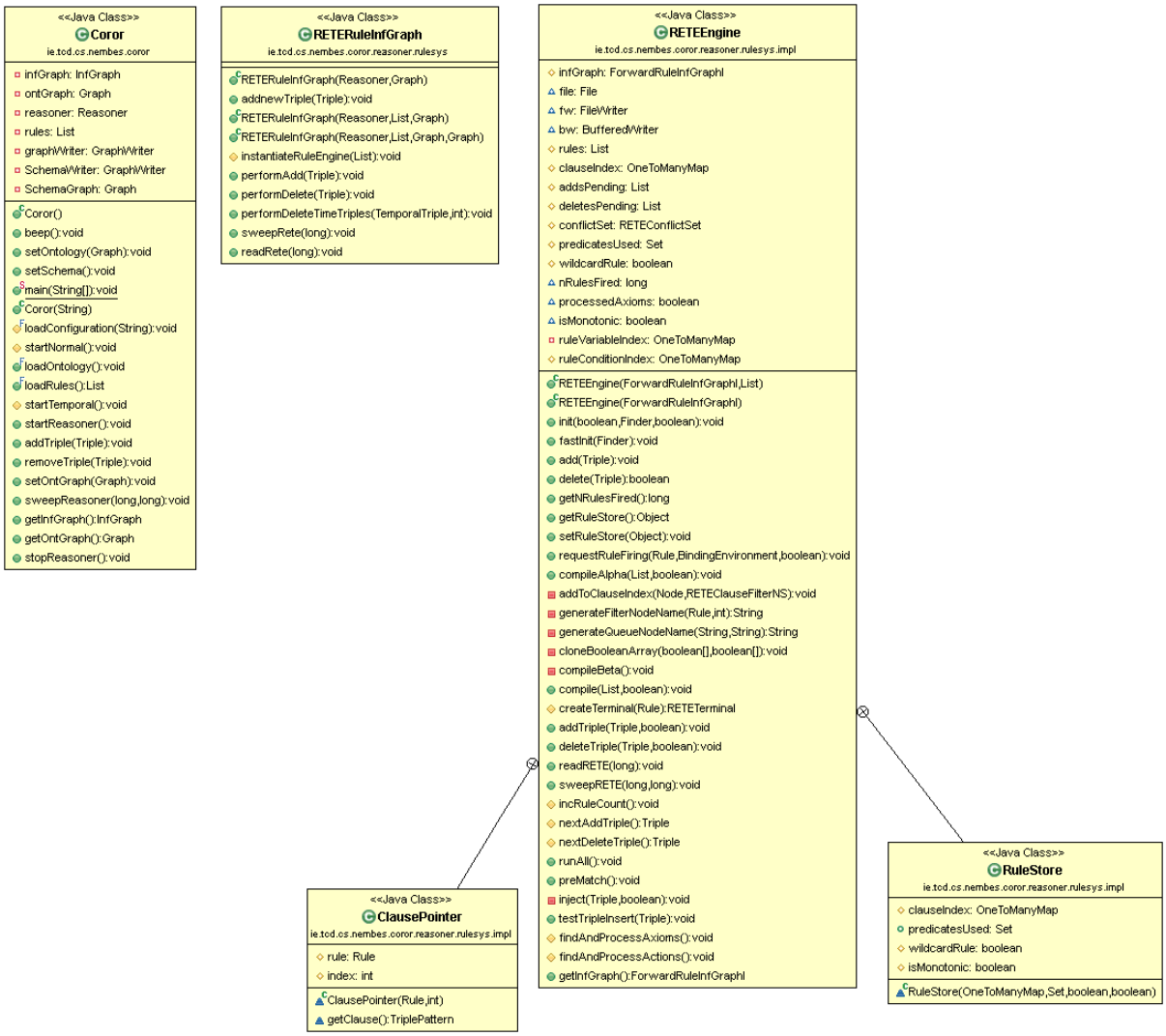
[rdfs12: (?v rdf:type rdfs:ContainerMembershipProperty) -> (?v rdfs:subPropertyOf rdfs:member)]

[rdfs13: (?v rdf:type rdfs:Datatype) -> (?v rdfs:subClassOf rdfs:Literal)]

pD*sv entailment

[rdf-svx: (?v owl:someValuesFrom ?w), (?v owl:onProperty ?p), (?u rdf:type ?v), notExistSomeValuesFromRestriction(?u, ?p, ?w), makeTemp(?b) -> (?u ?p ?b), (?b rdf:type ?w)]

Appendix C


«Java Class»
Coror
ie.tcd.cs.nembes.coror
infGraph: InfGraph
ontGraph: Graph
reasoner: Reasoner
rules: List
graphWriter: GraphWriter
SchemaWriter: GraphWriter
SchemaGraph: Graph
Coror()
beep():void
setOntology(Graph):void
setSchema():void
main(String[]):void
Coror(String)
loadConfiguration(String):void
startNormal():void
loadOntology():void
loadRules():List
startTemporal():void
startReasoner():void
addTriple(Triple):void
removeTriple(Triple):void
setOntGraph(Graph):void
sweepReasoner(long,long):void
getInfGraph():InfGraph
getOntGraph():Graph
stopReasoner():void
«Java Class»
RETERuleInfGraph
ie.tcd.cs.nembes.coror.reasoner.rulesys
RETERuleInfGraph(Reasoner,Graph)
addnewTriple(Triple):void
RETERuleInfGraph(Reasoner,List,Graph)
RETERuleInfGraph(Reasoner,List,Graph,Graph)
instantiateRuleEngine(List):void
performAdd(Triple):void
performDelete(Triple):void
performDeleteTimeTriples(TemporalTriple,int):void
sweepRete(long):void
readRete(long):void
«Java Class»
RETEEngine
ie.tcd.cs.nembes.coror.reasoner.rulesys.impl
infGraph: ForwardRuleInfGraphI
file: File
fw: FileWriter
bw: BufferedWriter
rules: List
clauseIndex: OneToManyMap
addsPending: List
deletesPending: List
conflictSet: RETEConflictSet
predicatesUsed: Set
wildcardRule: boolean
nRulesFired: long
processedAxioms: boolean
isMonotonic: boolean
ruleVariableIndex: OneToManyMap
ruleConditionIndex: OneToManyMap
RETEEngine(ForwardRuleInfGraphI,List)
RETEEngine(ForwardRuleInfGraphI)
init(boolean,Finder,boolean):void
fastInit(Finder):void
add(Triple):void
delete(Triple):boolean
getNRulesFired():long
getRuleStore():Object
setRuleStore(Object):void
requestRuleFiring(Rule,BindingEnvironment,boolean):void
compileAlpha(List,boolean):void
addToClauseIndex(Node,RETEClauseFilterNS):void
generateFilterNodeName(Rule,int):String
generateQueueNodeName(String,String):String
cloneBooleanArray(boolean[],boolean[]):void
compileBeta():void
compile(List,boolean):void
createTerminal(Rule):RETETerminal
addTriple(Triple,boolean):void
deleteTriple(Triple,boolean):void
readRETE(long):void
sweepRETE(long,long):void
incRuleCount():void
nextAddTriple():Triple
nextDeleteTriple():Triple
runAll():void
preMatch():void
inject(Triple,boolean):void
testTripleInsert(Triple):void
findAndProcessAxioms():void
findAndProcessActions():void
getInfGraph():ForwardRuleInfGraphI
«Java Class»
ClausePointer
ie.tcd.cs.nembes.coror.reasoner.rulesys.impl
rule: Rule
index: int
ClausePointer(Rule,int)
getClause():TriplePattern
«Java Class»
RuleStore
ie.tcd.cs.nembes.coror.reasoner.rulesys.impl
clauseIndex: OneToManyMap
predicatesUsed: Set
wildcardRule: boolean
isMonotonic: boolean
RuleStore(OneToManyMap,Set,boolean,boolean)