Slider: an Efficient Incremental Reasoner

Jules Chevalier, Julien Subercaze, Christophe Gravier, Frederique Laforest

Slider: an Efficient Incremental Reasoner

Jules Chevalier, Julien Subercaze, Christophe Gravier, Frédérique Laforest
Université de Lyon, F-42023, Saint-Etienne, France,
CNRS, UMR5516, Laboratoire Hubert Curien, F-42000, Saint-Etienne, France,
Université de Saint-Etienne, Jean Monnet, F-42000, Saint-Etienne, France.
firstname.name@univ-st-etienne.fr

ABSTRACT

The Semantic Web has gained substantial momentum over the last decade. It contributes to the manifestation of knowledge from data, and leverages implicit knowledge through reasoning algorithms. The main drawbacks of current reasoning methods over ontologies are two-fold: first they struggle to provide scalability for large datasets, and second, the batch processing reasoners who provide the best scalability so far are unable to infer knowledge from evolving data. We contribute to solving these problems by introducing Slider, an efficient incremental reasoner. Slider goes a significant step beyond existing system, including i) performance, by more than a 70% improvement in average compared to the fastest reasoner available to the best of our knowledge, and ii) inferences on streams of semantic data, by using intrinsic features that are themselves streams-oriented. Slider is fragment agnostic and conceived to handle expanding data with a growing background knowledge base. It natively supports ρdf and RDFS, and its architecture allows to extend it to more complex fragments with a minimal effort. In this demo a web-based interface allows the users to visualize the internal behaviour of Slider during the inference, to better understand its design and principles.

Keywords

Incremental Reasoning; Streamed Reasoning; Web of Data

1. INTRODUCTION

Data available on the Web can be represented through diverse range of formats. The Semantic Web movement enriches the available information by providing a stack of technologies – the core of which is the Resource Description Framework (RDF) for publishing data in machine-readable format. RDF data is represented as a set of triples of the form $<subject, predicate, object>$. Apart from the explicit data mapping, the integration of assertional data (i.e., instance data) with terminological data (i.e., structural data) described in ontologies permits deductive reasoning – i.e., inferring implicit knowledge. Although, various reasoning techniques has been described in literature but most popular of them is the rule-based reasoning – where each rule consist of i) an 'antecedent': a clause that allows the rule to be executed and ii) a 'consequent': the statements that can be inferred from the data that matched the antecedent clause. For instance,
`If` $<X, subClassOf, Y>$ `and` $<Y, subClassOf, Z>$
`Then` $<X, subClassOf, Z>$.
Reasoning can be described using a subsets of RDFS [5] and OWL standards [14] – also described as fragments. These fragments are set of rules with time complexity ranging from P to NEXTPTIME, and even undecideable for OWL2 Full [20].

Reasoning is inherently a complex process, and while there exists a large body of work in the area of reasoning algorithms and systems that work and scale well in confined environment [10, 19]; the distributed and dynamic nature of Web create new challenges for reasoning. This calls for new techniques to replace batch processing – where the arrival of new data initiate the reasoning process form the start – to incremental reasoning [2]. Thus to handle new data as soon as it arrive, without re-inferring the previously inferred knowledge.

The two main approaches for reasoning as discussed in literature includes, backward-chaining and forward-chaining. The first approach only uses the rules defined in the query to reason incoming triples at query time, while the second approach implements the materialisation process on the entire triple store. Both of these techniques have their pros and cons, in particular – backward-chaining suffers from more complex query evaluation that adversely affects performance and scalability but is suitable for more frequently changing knowledge bases, while forward-chaining enables scalability and very efficient responses at query time, but at the cost of an expensive up front closure computation. We choose forward-chaining (materialisation) to avoid high query response times and high resource usage that incurs in query-rewriting for backward-chaining – in order to conform to the dynamic and real-time nature of Web data.

Several solutions have been proposed to optimise incremental materialisation of ontologies. [8] proposed technique maintains classification of ontologies as they evolve, and provides encouraging results. However, its not a viable solution in case of static hierarchy of ontologies- i.e., if hierarchy is not affected by modification. Moreover, it is not adapted for ontologies with a high number of nominal. [12] handles both

.

addition and deletion in the setting of incremental classification. It is however limited to the classification on the TBox, and dedicated to a specific ruleset.

The major drawbacks that state-of-the art approaches suffer from is the inability to deal with complex ontologies and are not tailored to deal with large amount of dynamic RDF data and particularly large A-Boxes.

To overcome these drawbacks, we introduce Slider, an efficient reasoner to perform forward-chaining incremental reasoning and strikes a balance between large pre-processing cost of materialization and complexity of pure backward-chaining reasoning. Its core features that stands it apart from the previous approaches are as following.

Parallel and Scalable Execution: We implement a parallel and scalable method to perform incremental reasoning. Each inference rule is mapped to an independent module. These modules receive indented triples according to defined rules and later distribute triples to other modules for further processing. Additionally, we refine our algorithm to avoid bottlenecks and capitalise on scalability by allowing multiple instances of same rule to run in parallel – in order to further enhance the performance of the reasoner. Moreover, we implement efficient techniques for synchronized access of triple store and use buffers (blocking queues) to handle the explosion of inferred statements and incoming triples. This avoids any overheads and certifies the completeness of the reasoning process.

Duplicates Limitation: Reasoning on equivalent statements could result in a massive amount of duplicated data that causes decrease in performance. Thus, to avoid such situation we use the vertical partitioning approach along-with multiple indexing (on predicates, subjects and objects) technique. This limits the production of duplicates and avoids any unnecessary computation.

Data Stream Support: The dynamic nature of Web data results in data streams [9], which requires incremental execution of reasoning process. Due to the parallel nature of our architecture, Slider can handle both dynamic triple streams and static triples set. This, further allows the parallelisation of parsing and reasoning process on multiple data sources at the same time – processing data as soon as it is published.

Fragment's Customization: Reasoning is usually considered as once-off rule processing task, where rules are known a-priori. However, the dynamicity of data requires to migrate from task-specific systems. Slider natively supports both RDF and ρdf [16] fragments, and its architecture allows it to be further extended to any other fragments.

Slider outperforms existing implementations by 70% on average. We use OWLIM-SE (used in industry and the fastest available reasoner to the best of our knowledge) as a baseline system for comparative measurements. Our experimental evaluations prove that Slider outperforms OWLIM-SE by 106.86% on ρdf and 36.08% on RDFS. The source code of Slider is publicly available here: `https://github.com/juleschevalier/slider`.

Our demo system presents the reasoning process of Slider through a web-based interface and covers three main areas: i) Tuning reasoning process through various parameters including buffer size, buffer timeouts and rulesets. ii) Visualisation of each module behaviour and inputs. iii) Visualisation of synthetic inferred data and retrieval of original ontology.

2. SYSTEM ARCHITECTURE

In this section, we provide an overall description of the architecture of Slider. Our decentralised system consists of a set of autonomous modules – where each rule is mapped on a distinct module. A single triple store is shared by various concurrent modules in a synchronised manner and to ensure fast data access, triples are stored in memory.

Figure 1 shows the high-level architecture of the system with three inference rules R_1, R_2 and R_3. Incoming triples are send to both triple store and buffer of certain module – where each module accepts the triples according to configured rules' predicates. Once the buffer exceeds the configured size or timeouts, it initiates a new instance of a module that applies rules on buffered triples and relevant triples stored in the triple store. These newly created instances are managed by the thread pool for load distribution and scalability of the system. The distributor collects the resulted inferred data to be used as an input, and identifies the modules that would require the resulted data for their inference process – thus ensuring the completeness. For instance, the result of R_1 is used by R_2, R_3 and R_1 itself. Distributors and buffers play an important role in the architecture, specifically buffer orchestrates the load distribution of triples and instance creation of rule modules – as new instance for each triple can exhaust CPU resources. The core functionality of the system's components is as following.

Input Manager: It receives new triples and register them into a dictionary that maps the expensive URIs (as they introduce overheads during comparison computation) to `Longs`. Mapped triples are finally pushed into the triple store. Multiple instances of input manager allows to retrieve data from various sources.

Buffers: Each rule module is assigned with a buffer that is in-charge of collecting triples from input manager – by comparing the predicate values of configured rules. Once the buffer is full or in-case of timeouts, it triggers a new instance of rule module that applies rules on buffered triples.

Rule Modules: Rule modules are configured with a set of rules to be executed on incoming triples. It implements the forward-chaining reasoning by employing the input triples and relevant triples from the triple store.

Thread Pool: This component is responsible for: i) Receiving a new instance of rule module and implement pooling techniques for efficient resource usage, ii) Passing the inferred result to distributor. It pools and runs each instance of rule module on available resource for load balancing and allows asynchronous execution of instances to avoid overheads. Thus, ensuring the scalability of the system.

Distributors: It involves in three main tasks: i) Collecting inferred triples from rule modules ii) Adding inferred triples to triple store iii) Dispatching relevant inferred triples to relevant set of rules. Each distributor has a list of buffers that should receive newly inferred triples. This list is dynamically generated during reasoner initialisation by employing a rules dependency graph, as discussed later.

2.1 Rules Inference

The forward-chaining incremental reasoning is reported in pseudocode in Algorithm 1. It specifies how a rule, e.g. `CAX-SCO` (table 7 from [15]), effectively runs.

The algorithm searches for $<c_1$,`subClassOf`, $c_2>$ in the triple store and for $<x$,`type`, $c_1>$ from incoming triples, and vice versa. For each couple ($<c_1$,`subClassOf`, $c_2>$,

$<x$,type, $c_1>$) found, it creates a new one $<x$,type, $c_2>$.

Slider implements the rules from both ρdf and RDFS fragments and also support the addition of any new custom rules through Java interfaces.

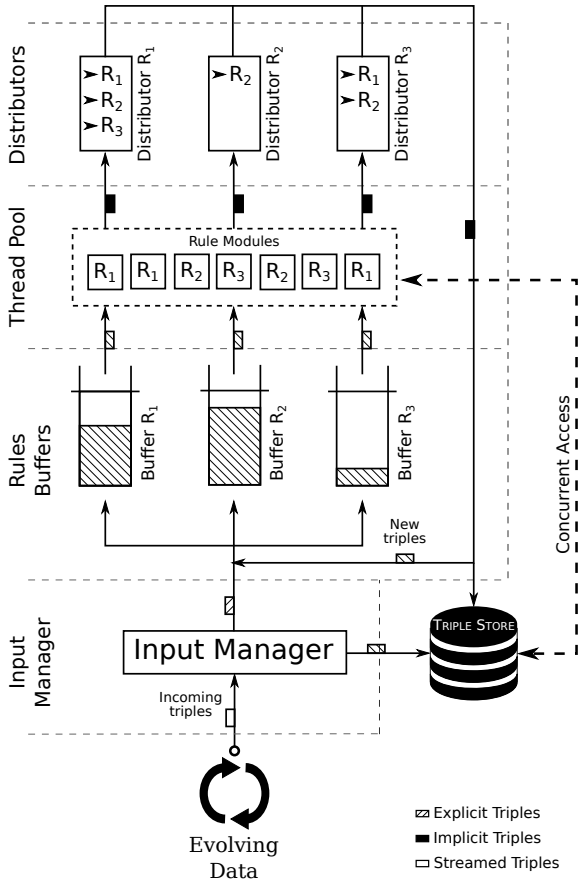

Figure 1: Global architecture of Slider

Algorithm 1 cax-sco

Require: tripleStore, newTriples, outputTriples

```
for all triple1 in TripleStore with predicate subClassOf do
  for all triple2 in newTriples with predicate type do
    if triple1.subject = triple2.object then
      output ← (triple2.subject,type,triple1.object)
      outputTriples ← outputTriples ∪ {output}
    end if
  end for
end for

for all triple1 in newTriples with predicate subClassOf do
  for all triple2 in TripleStore with predicate type do
    if triple1.subject = triple2.object then
      output ← (triple2.subject,type,triple1.object)
      outputTriples ← outputTriples ∪ {output}
    end if
  end for
end for
```

2.2 Triple Store

Triple store is another core component of the reasoning process – as good performance of the system relies on its ability to quickly retrieve a given pattern. In order to achieve high performance Slider uses a vertical partitioning approach as discussed in [1] – where triples are first indexed by predicates, later by subjects and finally by objects. We use `HashMaps` of `MulitMaps` from Guava library[1] to implement triples' indexing. The concurrency of the triple store is handled by using `ReentrantReadWriteLock`, which provides both read and write (during addition of new triples) locks. This two-layered locking mechanism not only ensures concurrency, but also allows parallel access to the triple store. The `HashMap` structure of indexing scheme ensures the duplicate management in triple store. Distributors in-charge of dispatching new triples to the buffers use this feature to exclude duplicates. Furthermore, after adding inferred triples in the triple store only distinct triples are send to the buffers.

If we consider the entire ruleset for OWL [15] (especially tables 4–8 and 9), all the associated rules processing require either to walk the entire set of triples (example `eq-ref` from Table 4 in [15]); or to access the triples by predicate first (example: `eq-sym` from the same table). For this reason, triples are firstly indexed by predicate, then by subject and finally by object. This provides the best trade-off for near-optimal indexing for nearly all rules even in the most expressive OWL fragment.

2.3 Rules Dependency Graph

During the initialization process, Slider creates a list of dependent buffers for each rule – that is later utilise by the distributor to send the inferred triples to the corresponding buffers. To implement such functionality, Slider builds a *rules dependency graph*. It is a a directed graph, where edges represent the links (dependency) between the rules (vertices). For instance, if there is a directed edge between rule A and B then the output of rule A can be used by rule B. Figure 2 depicts the dependency graph for ρdf fragment. The rules (`PRP-SPO`, `PRP-RNG`, `PRP-DOM`) with universal input accept all kinds of triples. As according to the Figure 2, the directed edge from rule `SCM-SCO` to `CAX-SCO` depicts that output of first rule, a `subclassOf` relation can be used as an input for second rule. This dependency graph is used to create a list of dependency buffers for each distributor.

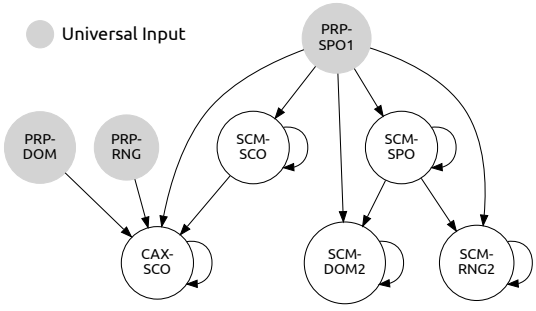

Figure 2: Rules dependency graph for ρdf

[1] `https://github.com/google/guava`

3. EXPERIMENTATIONS

In this section we describe the experimental details and comparison analysis of Slider with OWLIM-SE.

OWLIM-SE (Standard Edition) [3] is a semantic repository with reasoning features. It was setup to use the default ρdf and RDFS rulesets along-with the custom rule configuration available in OWLIM-SE for ρdf. Therefore, we use the same ruleset for inference process and it is strictly the same. As OWLIM-SE does not allows to separately compute the parsing and inference time, thus in our experiments, for both systems, the running times include both parsing and inferencing times.

Our experimental settings use a set of 13 ontologies divided in three categories. The first one contains generated ontologies, where we use Berlin SPARQL Benchmark (BSBM) [4] to generate five ontologies from 100,000 to 5 million triples. These ontologies shows Slider ability to handle high rates of data by producing smaller set of inferred triples during reasoning process.

The second category of ontologies only contains `subClassOf` relations, where Equation 1 details how a `subClassOf`n ontology is generated.

$$\begin{array}{l} < 1, type, Class > \\ < i, type, Class > \qquad i \in \{2, 3, \ldots, n\} \\ < i, subClassOf, (i-1) > \end{array} \qquad (1)$$

These ontologies are easy to generate but provide the utmost practical interest due to their complexity. The chain of n rules produce $O(n^2)$ unique triples, however commonly used iterative rules schemes produce $O(n^3)$ triples [19]. These ontologies are used to test and compare the Slider ability to handle duplicates.

The last category of ontologies contains the real-world ontologies: a Wikipedia based ontology, and the other based on WordNet [18].

These ontologies are representative of synthetic data (issued from BSBM benchmark generator tool), extensive closure computation[11] (chained subsomptions), and ontologies of practical interest (Wordnet and Wikipedia). This dataset contains more ontologies and of higher diversity than previous studies in the field [13, 17], and is publicly available on the Web using the link provided in this demo.

We ran our benchmark on on a standalone machine under Linux Ubuntu 12.04, with an AMD processor with 4 1.4GHz cores, and 16GB RAM.

Table 1 enumerates the ontologies, with the results of the benchmark. Figure 3 shows the comparison of inference time between OWLIM-SE and Slider, for both ρdf and RDFS. The results on `BSBM_5M` ontology have been omitted in Figure 3 for the sake of clarity.

These experiments show that Slider on average is 71.47% faster than OWLIM-SE, with a throughput up to 36,000 Triples/sec. It exhibits an interesting speed-up of 106.86% for ρdf and 36.08% for RDFS.

We believe that the outcome of our evaluation is very significant: Slider outperforms OWLIM-SE, a commercial product that itself outperforms Jena [7] and Sesame [6] native reasoners.

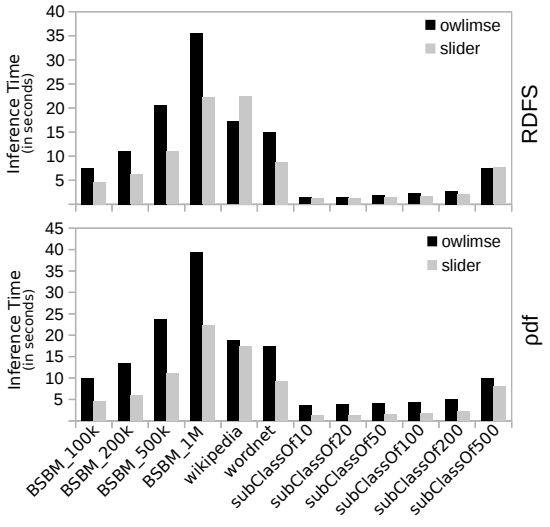

Figure 3: Inference time comparison between Slider and OWLIM-SE, on ρdf and RDFS(Lower is better)

		ρdf reasoning				RDFS reasoning			
Ontology	**Input**	**Inferred**	**OWLIM**	**Slider**	**Gain**	**Inferred**	**OWLIM**	**Slider**	**Gain**
BSBM_100k	99914	544	9.907s	**4.636s**	113.69%	33752	7.487s	**4.558s**	64.25%
BSBM_200k	200007	1102	13.338s	**6.059s**	120.12%	64492	11.064s	**6.198s**	78.52%
BSBM_500k	500037	4347	23.595s	**11.133s**	111.93%	157831	20.580s	**10.984s**	87.36%
BSBM_1M	1000000	8664	39.364s	**22.357s**	76.07%	304065	35.602s	**22.192s**	60.43%
BSBM_5M	5000000	43212	170.151s	**126.292s**	34.73%	1449107	160.699s	**127.037s**	26.50%
wikipedia	458369	191574	18.802s	**17.422s**	7.92%	555653	**17.186s**	22.443s	-23.42%
wordnet	473589	0	-	-	-	321888	15.075s	**8.828s**	70.77%
subClassOf10	20	36	3.507s	**1.209s**	190.05%	50	1.423s	**1.216s**	16.99%
subClassOf20	40	171	3.730s	**1.316s**	183.41%	195	1.536s	**1.330s**	15.53%
subClassOf50	100	1176	4.159s	**1.615s**	157.49%	1230	1.865s	**1.583s**	17.78%
subClassOf100	200	4851	4.397s	**1.827s**	140.60%	4955	2.242s	**1.805s**	24.21%
subClassOf200	400	19701	4.962s	**2.210s**	124.56%	19905	2.837s	**2.170s**	30.69%
subClassOf500	1000	124251	9.862s	**8.102s**	21.72%	124755	**7.584s**	7.625s	-0.54%
				Average	**106.86%**			Average	**36.08%**

Table 1: Benchmark results for Slider and OWLIM-SE inference on ρdf and RDFS

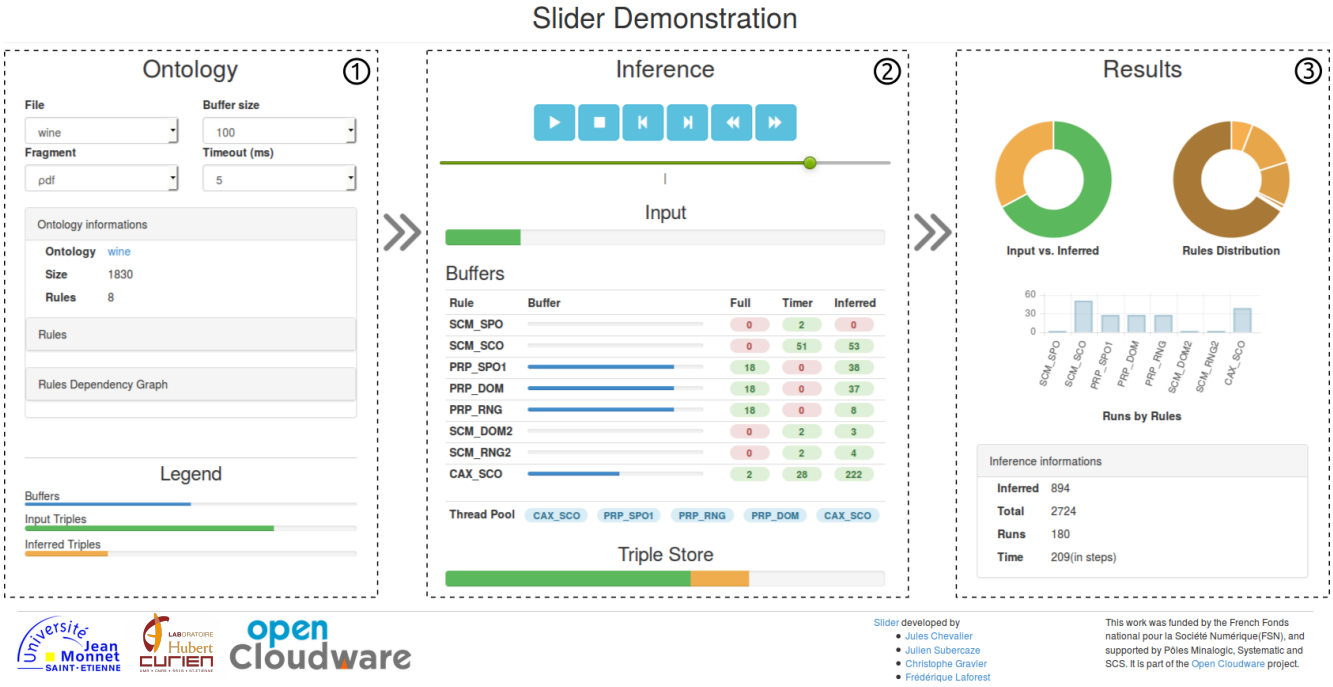

Figure 4: Demonstration Web Interface

4. DEMONSTRATION

The demonstration will allow attendees to visualize the internal behaviour of Slider during the inference. A comprehensive web GUI will enable users to edit Slider's parameters, to choose from a set of 11 ontologies, to interact with the inference process and to experience how the system actually performs. It allows the user to edit 24 configurations of the reasoner, and implement 264 different scenarios. For each one, we ran the reasoner and logged the state of all the modules of Slider at each step of the process. This allows the user, through an *inference player*, to pause the inference, to go backwards, and to replay any part of the inference. All the information on the inference process is presented to the users, and to provide them with the opportunity to appreciate the effect of each parameter.

Figure 4 is a screenshot of the demonstration. It is fully accessible at this link:
`http://demo-satin.telecom-st-etienne.fr/slider/`

The interface of the demonstration is composed of three sections, corresponding to the three following steps of the demonstration.

① **Setup**: To begin, the users can tune the parameters of the reasoner. They can choose the ontology among the 11 available ones. Then three additional parameters can be adjusted. The fragment (ρdf or RDFS), which defines the set of rules applied during the inference; the size of the buffers, which determines how many triples are needed to fire a new rule execution; and the timeout, which defines after how long a inactive buffer is forced to flush and throw a rule execution. The table *Informations*, located on the right of the panel, summarizes essential informations about the ontology. The ontologies can be downloaded following the links provided in this table. The definition of each rule of the chosen fragment is available on the bottom of the panel ①. The rules dependency graph follows this.

② **Run**: Once the configuration of the parameters is finished, users will be able to lunch the inference process. They will observe the internal modules behaviour during the reasoning process. The state of the input parsed, as well as the buffers and the triple store, are represented by progress bars. The input is progressively emptied, while the triples go through the buffers. For each buffer, three counters represent: i) the number of times it has been full (and so a new thread has been created), ii) the number of times the buffer has been forced to flush because of the timeout and iii) the amount of triples the rule associated to the buffer has inferred. Under the buffers, the thread pool is represented by the last five executed rules. Finally, on the bottom of the panel, the triple store is represented with a two-coloured progress bar, the green part representing the input triples, coming from the ontology, while the orange part represents the inferred triples.

To let the user see exactly what happened at every step, the entire process can be played, paused, accelerated and slowed down, played step by step, and a slider bar allow to scroll through each step.

③ **Summarize**: The last panel summarizes the informations about the inference: proportion of triples from the ontology compared with the triples inferred, distribution by rule of the triples inferred, and number of time each rule has run are presented in three charts. Miscellaneous informations on the *quality* of the inference and the impact of the parameters on it are shown in the table below. Number of rules executions, inferred triples, inference time, all these measures, coupled with the animation, help the user to evaluate the impact of the different parameters on the performance and on the behaviour of the reasoner.

5. CONCLUSION AND FUTURE WORK

In the frame of reasoning on evolving data, few proposals enable to continuously infer new knowledge as new explicit triple are sent to the reasoner. Most of the solutions limit the amount of data in the knowledge base by eliminating former triples.

Instead of firing a full inference at regular interval of time, we propose Slider, a reasoner that handles triples flows as the very core of its architecture. The triple store, built with the vertical partitioning, ensures fast triples retrieval and a minimal space occupation. To be fragment agnostic, Slider builds a rules dependency graph of the fragment it reasons over at initialization time. It then uses this information to plug the rules together, creating the route of the triples in the reasoner. This provides a high flexibility in its architecture. This design is strongly original towards traditional reasoning scheme like the famous RETE algorithm.

We evaluate Slider against 3 sets of ontologies : a first set containing generated ontologies, issued by BSBM benchmark tools, a second set specific to the worst-case reasoning on subsumption relationship, and finally real field ontologies. On all kinds of ontologies Slider significantly outperforms OWLIM-SE, an industrial-strength reasoner, by a factor of 71,47%.

For our future endeavours, we would like to focus on two main aspects of Slider. First, we will implement more complex inference rules, in order to implement reasoning over a more complex fragments. Second, we will implement a just-in-time optimisation of the rules execution's scheduling. Therefore, migrating from 'static' plans produced by traditional optimizers to the run-time dynamic plans will improves the Slider ability to adapt and be more reactive – i.e., learning from ontologies structures and previously executed runs.

6. ACKNOWLEDGMENTS

This work is supported by the OpenCloudware project. OpenCloudware is funded by the French FSN (Fond national pour la Société Numérique), and is supported by Pôles Minalogic, Systematic and SCS. We would also like to thank Syed Gillani for his careful help in proofreading this paper.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *PVLDB*, 2007.

[2] D. Barbieri, D. Braga, S. Ceri, E. Valle, and M. Grossniklaus. Incremental Reasoning on Streams and Rich Background Knowledge. In *The Semantic Web: Research and Applications*. 2010.

[3] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A Family of Scalable Semantic Repositories. *Semantic Web*, 2011.

[4] C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems*, 2009.

[5] D. Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF schema. 2004.

[6] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*. 2002.

[7] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *WWW*, 2004.

[8] B. Cuenca Grau, C. Halaschek-Wiener, and Y. Kazakov. History Matters: Incremental Ontology Reasoning Using Modules. In *The Semantic Web*. 2007.

[9] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *SIGMOD*, 2005.

[10] J. Hoeksema and S. Kotoulas. High-performance distributed stream reasoning using s4. In *ISWC*, 2011.

[11] H. Jagadish, R. Agrawal, and L. Ness. A study of transitive closure as a recursion mechanism. In *SIGMOD*, 1987.

[12] Y. Kazakov and P. Klinov. Incremental Reasoning in OWL EL without Bookkeeping. In *ISWC*. 2013.

[13] J. Liagouris and M. Terrovitis. Efficient Identification of Implicit Facts in Incomplete OWL2-EL Knowledge Bases. *PVLDB*, 2014.

[14] D. L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C Recommendation*, 2004.

[15] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. Owl 2 web ontology language: Profiles. `http://www.w3.org/TR/owl2-profiles/`, 2009.

[16] S. Munoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In *The Semantic Web: Research and Applications*. 2007.

[17] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB*.

[18] V. Snasel, P. Moravec, and J. Pokorny. WordNet ontology based model for web retrieval. In *Web Information Retrieval and Integration*, 2005.

[19] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In *The Semantic Web: Research and Applications*. 2010.

[20] W3C. OWL 2 Language Primer. `http://www.w3.org/TR/owl2-primer/`, 2012.