Towards Scalable Non-monotonic Stream Reasoning via Input Dependency Analysis

Thu-Le Pham
Insight Centre for Data Analytics
National University of Ireland, Galway
IDA Bussiness Park,
Lower Dangan, Galway, Ireland
Email: thule.pham@insight-centre.org

Alessandra Mileo
Insight Centre for Data Analytics
Dublin City University
Glasnevin, Dublin 9
Dublin, Ireland
Email: alessandra.mileo@insight-centre.org

Muhammad Intizar Ali
Insight Centre for Data Analytics
National University of Ireland, Galway
IDA Bussiness Park,
Lower Dangan, Galway, Ireland
Email: ali.intizar@insight-centre.org

Abstract—**Stream reasoning is an emerging research area focused on providing continuous reasoning solutions for data streams. The high expressiveness of non-monotonic reasoning enables complex decision making by managing defaults, common-sense, preferences, recursion, and non-determinism, but it is computationally intensive. The exponential growth in the availability of streaming data on the Web has seriously hindered the applicability of state-of-the-art non-monotonic reasoners to be applied to streaming information in a scalable way.**

In this paper, we address the issue of scalability for non-monotonic stream reasoning based on Answer Set Programming (ASP) - an expressive reasoning approach based on disjunctive logic Datalog with negation under the stable model semantics, by analyzing input dependency. We introduce an input dependency graph to represent the relationships between input events based on the structure of a given logical rule set. The input dependency graph allows us to dynamically configure the streaming window size in order to maximise the scalability of the non-monotonic reasoner. We conduct an experimental evaluation to demonstrate the effectiveness and ability of our proposed approach in improving the scalability of disjunctive logic programming with ASP in dynamic environments.

I. Introduction

The variety of real-world applications in the IoT space requires reasoning capabilities that can handle incomplete and potentially inconsistent input and extract knowledge from it to facilitate decision support. While semantic technologies for handling data streams cannot exhibit complex reasoning capabilities, logic-based non-monotonic reasoning approaches can be quite costly in terms of efficiency. Declarative Web stream reasoning is an emerging research area which explores advances in stream processing technologies for representing and processing data streams on the one hand, and non-monotonic reasoning approaches for performing complex rule-based inference in changing environments on the other hand. This combination is based on the principle of having a 2-tier approach where: i) a stream processor is used to filter semantic data elements, and ii) a non-monotonic reasoner is used for computationally intensive tasks. Such a combined approach can help to improve the scalability of complex reasoning over semantic streams since the size of input of the non-monotonic reasoner is reduced by the stream processor.

Current declarative stream reasoning systems, like ASR [8], EP-SPARQL [1], and StreamRule [16], support non-monotonic reasoning over data streams. In particular, ASR uses the DLVhex solver [9], EP-SPARQL uses ETALIS [2] which is implemented in Prolog, and StreamRule uses the Clingo solver [10] as a subprocess to infer new knowledge from data streams and a given rule set. In order to enable these solvers, a middle layer is implemented for transformation between data formats. For example, the StreamRule system intercepts the output RDF stream query results filtered by CQELS [15] and translates them into Answer Set Programming (ASP) syntax before streaming them into Clingo. Given the data transformation overhead, performance of the reasoning subprocess should be measured by not only the processing time of the solver but also the time required for data transformation. Moreover, the reasoning component needs to return results faster than when new input arrives in order to maintain the stability of the whole system. This requires optimization techniques that can speed up the processing.

Recently, authors in [12] have focused on studying heuristics in order to achieve a better performance of the reasoning subprocess in StreamRule (Figure 1). They have analyzed the correlation between the streaming rate and the (tuple-based) window size in order to optimise the expressivity vs. scalability trade-off in dynamic environments. Their approach suggests to partition data into chunks and process them sequentially. The approach seems to improve StreamRule performance, but their naive solution is based on the assumption that input data in the window is independent, which is rarely a case. Moreover, partitioning data randomly in general decreases the accuracy of the answers. To overcome this issue, the data partitioning process needs to take into account the dependencies among data items.

Consider the reasoning subprocess in StreamRule with the declarative encoding of the input program P (a set of rules) in ASP syntax. In this paper, we study the data partitioning process which leverages the data dependency in order to: (i) enable parallelism in the reasoning subprocess of StreamRule, and (ii) maximise the accuracy of the answers. In particular, we propose the construction of an input dependency graph by studying the structure of P. This graph represents how data items in a window relate to each other. Then we identify a partitioning plan to guide the data partitioning process at run-time. We extend the architecture of StreamRule with the parallel reasoning subprocess. We conduct an experimental evaluation to show that partitioning input while considering

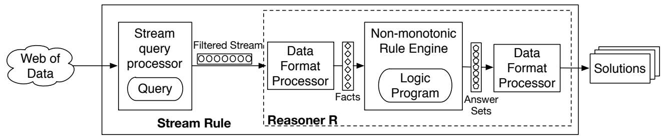

Fig. 1: Conceptual architecture of StreamRule

their dependencies can improve the reasoning performance and accuracy of the answers.

For the rest of the paper, we use *solver* to mention the *non-monotonic rule engine* and *reasoner* R to refer to the subprocess in StreamRule which includes the *solver* and the *data format processor* (the dashed box in Figure 1). The *logic program* (or *program*) P is a set of rules in ASP. $pre(P)$ denotes the set of predicates in P. $inpre(P)$ denotes predicates of input data items of P ($inpre(P) \subseteq pre(P)$). The reasoner R receives the input data items from the stream query processor. An *input window* (or *window)*, W, is a set of input data items that the reasoner R processes per computation. From the logical point of view, the data items in W can be called as *ground atoms*. $pre(W)$ defines the set of predicates of ground atoms in W. Therefore, $pre(W) \subseteq inpre(P)$.

II. Input Dependency Analysis

In this section, we discuss the problem of analysing dependency of input elements in a window W of the reasoner R with respect to a set of ASP rules in a program P. We first introduce a motivating example to highlight the critical aspects of analysing dependency. Thereafter, we propose the input dependency graph that shows how input data items in W relates to each other with respect to P.

A. Motivating Example

Consider the following example: A city manager wants to know real-time traffic events happening in the city in order to react accordingly. For this reason, he wants to develop an instance of the StreamRule system that detects events of interest, for example, *traffic_jam* and *car_fire* as defined in the logic program P in Listing 1. P is given as input to the solver of StreamRule, together with $inpre(P)$ = {*average_speed, car_number, traffic_light, car_in_smoke, car_speed, car_location*}. The reasoner R is triggered whenever a new input window W arrives from the stream processor.

```
(r1) very_slow_speed(X) :- average_speed(X,Y), Y < 20.
(r2) many_cars(X) :- car_number(X,Y), Y > 40.
(r3) traffic_jam(X) :- very_slow_speed(X), many_cars(X), not traffic_light(X).
(r4) car_fire(X) :- car_in_smoke(C,high), car_speed(C,0), car_location(C,X).
(r5) give_notification(X) :- traffic_jam(X).
(r6) give_notification(X) :- car_fire(X).
```

Listing 1: Rules for detecting events

Assume at time t, an input window (in ASP format) arrives as follows: W = {*average_speed(newcastle, 10), car_number(newcastle, 55), traffic_light(newcastle), car_in_smoke(car1,high), car_speed(car1,0), car_location(car1,dangan)*}. In order to process W faster, partitioning W randomly into chunks applied as in [12] could generate wrong results. For example W_1 = {*average_speed(newcastle, 10), car_number(newcastle, 55), car_in_smoke(car1,high)*} and W_2 = {*traffic_light(newcastle), car_speed(car1,0), car_location(car1,dangan)*}. Reasoning in parallel over these two input partitions produces as a result the event *traffic_jam(newscastle)* and the action *give_notification(newcastle)* is triggered, which is not correct. The accurate answer is the event *car_fire(dangan)* detected and the notification about the *dangan* road segment. Therefore, the partitioning process should consider the relations between ground atoms in the window.

B. Input Dependency Graph

Consider a logic program P and a set of input predicates $inpre(P)$ for P. In this section, we introduce a dependency graph that represents how predicates in $inpre(P)$ relate to each other by analysing the structure of P.

The concept of *dependency graph* has been widely used in ASP as a tool to analyse the structure of non-ground answer set programs [6], [18]. It has been efficiently used in a parallel instantiation[1] algorithm that generates a much smaller ground program equivalent to a given logic program. As defined in [6], the dependency graph G is a directed graph where nodes are *IDB (intensional database)* predicates and arcs show the relationship between a positive IDB predicate in the body with a predicate in the head of a rule. This graph divides the input program P into subprograms, according to the dependencies among the IDB predicates of P, and identifies which of them can be evaluated in parallel. However, in this paper, we are not partitioning the logic program. We are focusing on partitioning the input and evaluating each partition in parallel with a copy of the whole program P. The input predicates can be either IDB or *EDB (extensional database)* predicates. Therefore, besides the dependencies among IDB predicates defined in the dependency graph, other relationships should be taken into account, such as between two EDB predicates, or between an IDB predicate and an EDB predicate.

According to the above argument, we first define an *extended dependency graph* from the definition in [6]. This graph shows different types of dependency among predicates in P by considering: i) the (transitive) relation between two predicates (both IDB and EDB) in the body of a rule, ii) both positive and negative literals. Based on this extended dependency graph, we introduce the *input dependency graph* of P with respect to $inpre(P)$. This input dependency graph describes how predicates in $inpre(P)$ depend on each other.

Definition 1: Let P be a logic program. The *extended dependency graph* of P is a graph $G_P = \langle N_P, E_P \rangle$, where:

i) N_P is a set of nodes, where each node represents a predicate in $pre(P)$.
ii) $E_P = E_{P_1} \cup E_{P_2}$, where:
   a) E_{P_1} contains undirected edges $e_u = (p_u, q_u)$ if p_u and q_u occur in the body of a rule[2] r in P. Moreover, $(p_u, p_u) \in E_{P_1}$ if $p_u \in body^-(r)$[3].
   b) E_{P_2} contains directed edges $e_d = \langle p_d, q_d \rangle$ if q_d occurs in the head of r and p_d occurs in the body of r.

[1]The computation of most ASP systems follows a two-phase approach: an instantiation (or grounding) phase generates a variable-free program which is then evaluated by propositional algorithms in the solving phase.

[2]In ASP, a rule has form: $q_1 \vee ... \vee q_n \leftarrow p_1, ..., p_k, not\, p_{k+1}, ..., not\, p_m$ where $q_1, ..., q_n, p_1, ..., p_m$ are atoms and $n \geq 0, m \geq k \geq 0$.

[3]$body(r) = \{p_1, ..., p_k, not\, p_{k+1}, ..., not\, p_m\}$ is the body of r. $body^-(r)$ (respectively, $body^+(r)$) denotes the set of atoms occurring negatively (respectively, positively) in $body(r)$.

Note that p_u, q_u, p_d, q_d can be either a positive or a negative literal.

Example 1: Consider the program P in Listing 1. The extended dependency graph G_P illustrated in Figure 2 represents different relations among predicates in P including directed and undirected edges.

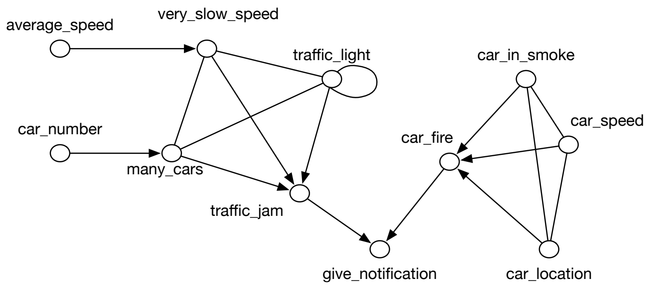

Fig. 2: Extended dependency graph G_P

Definition 2: Let P be a logic program and $inpre(P)$ be a set of input predicates of P. The *input dependency graph* of P with respect to $inpre(P)$ is an undirected graph $G_P^{inpre(P)} = \langle N_P^{inpre(P)}, E_P^{inpre(P)} \rangle$, where $N_P^{inpre(P)} \subset N_P$ is a set of nodes and $E_P^{inpre(P)}$ is a set of edges. $N_P^{inpre(P)}$ contains a node for each predicate in $inpre(P)$, and $\forall p, q \in N_P^{inpre(P)}$, $(p, q) \in E_P^{inpre(P)}$ if one of the following conditions is satisfied:

i) $(p, q) \in E_{P_1}$
ii) There is a sequence of nodes $p_1, p_2, ..., p_{n-1}, p_n$ ($n > 1, p_1 = p, p_n = q$) such that $\exists! i \in [1, n]$, $(p_i, p_{i+1}) \in E_{P_1}$ and there are two directed paths[4], one is from p_1 to p_i, the other is from p_n to p_{i+1}.
iii) $p = q$ and $\exists u \in N_P, (u, u) \in E_{P_1}, \langle p, u \rangle \in E_{P_2}$

Example 2: Consider the extended dependency graph G_P in *Example 1* with the input predicates $inpre(P)$ = {*average_speed, car_number, traffic_light, car_in_smoke, car_speed, car_location*}. The input dependency graph $G_P^{inpre(P)}$ is shown in Figure 3.

Definition 3: Predicates $p, q \in inpre(P)$ *depend on each other* if there is an edge (p,q) in the input dependency graph $G_P^{inpre(P)}$.

In Definition 2, the first two conditions show that for any two different predicates $p, q \in inpre(P)$ connected by an edge in the graph $G_P^{inpre(P)}$, they depend on each other. It means that they can contribute to infer a new fact by firing a single rule (condition (i)) or multiple rules (condition (ii)). The third one identifies a self-loop[5] of any predicate if its father node has a self-loop. The self-loop of a predicate describes the dependencies among ground atoms of that predicate. For example, in Figure 3, the subgraph on the right illustrates the dependencies among predicates *car_in_smoke*, *car_speed*, and *car_location* which held by condition (i), while the subgraph on the left shows the dependencies among *average_speed*, *traffic_light*, and *car_number* which derived by condition (ii). The self-loop of the predicate *traffic_light* means that all ground atoms of *traffic_light* depend on each other. Dependent predicates (or dependent ground atoms) need to be processed together in order to guarantee that rules in P are fired properly and therefore, reduce the number of incorrect answers.

[4]A directed path from node p_1 to node p_n is a sequence of nodes $p_1, p_2, ..., p_n$ such that $\langle p_1, q_2 \rangle$, $\langle p_2, q_3 \rangle$, ..., $\langle p_{n-1}, q_n \rangle \in E_{P_2}$

[5]A self-loop is an edge that connects a vertex to itself.

The input dependency graph $G_P^{inpre(P)}$ that is not connected[6] induces naturally a subdivision of $inpre(P)$ into several *connected components*[7] (or *components*). For instance, $G_P^{inpre(P)}$ in Figure 3 decomposes $inpre(P)$ into two components {*average_speed*, *traffic_light*, *car_number*} and {*car_in_smoke*, *car_speed*, *car_location*}. These components are used in the partitioning process for splitting ground atoms in a window on-the-fly.

However, there are some cases where the input dependency graph $G_P^{inpre(P)}$ is connected so that it is not straightforward to create connected components of $inpre(P)$. For example, consider the logic program P' which includes P in Listing 1 and this following rule:

```
(r7) traffic_jam(X) :- car_fire(X), many_cars(X).
```

Assume that $inpre(P') = inpre(P)$. The input dependency graph $G_{P'}^{inpre(P')}$ is shown in Figure 4. This graph is connected. Optimising the performance of the reasoner R by data partitioning approach can not be applied if the input dependency graph can not be decomposed. To cope with this issue, we introduce the *decomposing process* to divide the graph by duplicating some common nodes. This process has three main steps: (1) using the modularity algorithm [4] to decompose the input dependency graph into disjoint subgraphs (also called *communities* in graph theory); (2) for any two communities C_1 and C_2, identify a set of the nodes $exnodes(C_1)$ in C_1 which have links to nodes in C_2 (similarly $exnodes(C_2)$ for C_2); (3) between $exnodes(C_1)$ and $exnodes(C_2)$, the set of nodes are chosen to duplicate is the one which has smaller cardinality and these nodes which are called *duplicated nodes* will belong to both C_1 and C_2. The output of this process is a set of mappings from predicates to their communities (we call this output *partitioning plan*).

Example 3: Consider the input dependency graph $G_{P'}^{inpre(P')}$ in Figure 4. Step 1 of the duplication process decomposes the graph into two communities[8] C_1 = {*traffic_light, average_speed, car_number*} and C_2 = {*car_in_smoke, car_speed, car_location*}. Step 2 identifies $exnodes(C_1)$ = {*car_number*} and $exnodes(C_2)$ = {*car_in_smoke, car_speed, car_location*}. Finally, step 3 chooses {*car_number*} to duplicate. The output of the duplication process has two components where the duplicated node is *car_number* (see Figure 5).

III. Extended StreamRule

The StreamRule framework extended with the partitioning process in the reasoning layer is shown in Figure 6. The extension consists of the *partitioning handler* and the *combining handler*. The partitioning handler splits an input window W coming from the stream query processor into several

[6]An undirected graph is connected if for every pair of vertices, there is a path in the graph between those vertices.

[7]A connected component of an undirected graph is a maximal connected subgraph of the graph.

[8]We use the resolution[14] = 1.0 in the modularity algorithm.

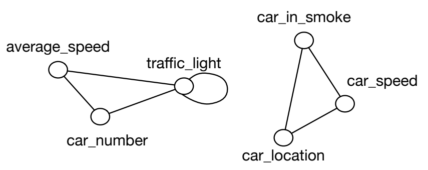

Fig. 3: Input dependency graph $G_P^{inpre(P)}$

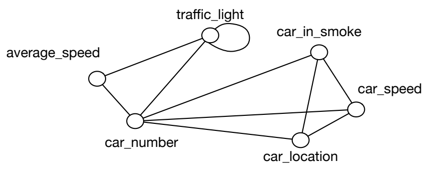

Fig. 4: Input dependency graph $G_{P'}^{inpre(P')}$

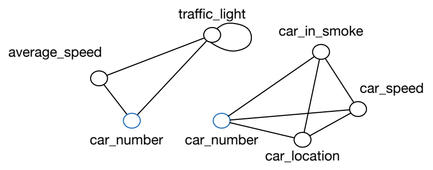

Fig. 5: Output of the decomposing process for $G_{P'}^{inpre(P')}$

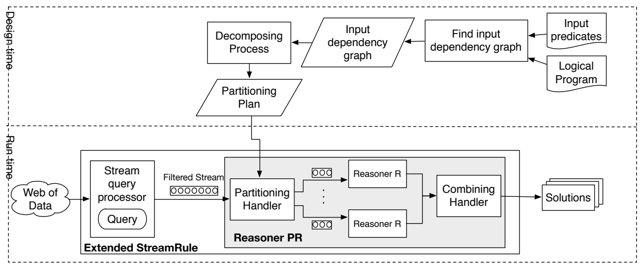

Fig. 6: The Extended StreamRule

sub-windows taking into account the input dependency. The combining handler combines outputs from parallel reasoners. For the realization of the partitioning process, the analysis of input dependency is made available within the framework at first at design time. To achieve this, a logic program and a set of input predicates are given in advance in order to build an input dependency graph as defined in Definition 2. Then the *duplication process* (see Section II-B) builds a partitioning plan by decomposing this graph into several components with their duplicated predicates.

The partitioning handler. At run-time, the partitioning handler starts to split an input window on-the-fly by using the partitioning plan provided at design-time. Algorithm 1 shows the partitioning process. First, the *group()* method classifies items in the window by their predicates (Line 3). For each group of items, the algorithm identifies a set of communities' IDs that group belongs to based on the partitioning plan (Line 5). Finally, it adds that group into the proper partitions corresponding to those IDs.

The combining handler. The output of the reasoner R in StreamRule is non-deterministic since R uses the ASP solver for reasoning. Therefore, R may return different results for the same input. Given a logic program P and an input window W, the answers provided by R over P and W (notated as $Ans_P(W)$) are computed as:

$$Ans_P(W) = \{ \bigcup_{i=1}^{n} ans_i : ans_i \in Ans_P(W_i) \}$$

Where W_i $(i = 1..n)$ is a partition of W provided by the partitioning handler.

To close this section, we introduce the concepts of latency and accuracy that are used for the evaluation of the *reasoner* PR (the grey box in Figure 6). The *reasoning latency* is the time required for the reasoner PR to process an input window. We define $Ans_P^R(W)$ and $Ans_P^{PR}(W)$ as the answers provided by the reasoner R and PR respectively. Generally, the accuracy is the ratio between the number of elements of $Ans_P^{PR}(W)$ that are also in $Ans_P^R(W)$ and the total number of the elements

Algorithm 1 Partitioning method

Input: a partitioning plan ρ and an input window W
Output: sub-windows of W

```
1:  procedure PARTITION($\rho, W$)
2:      $Partitions \leftarrow [\ ]$;
3:      $G \leftarrow group(W)$;
4:      for $g \in G$ do
5:          $C \leftarrow findCommunities(\rho, g.predicate)$;
6:          for $c \in C$ do
7:              Add $g.items$ into $Partitions[c]$;
8:          end for
9:      end for
10:     return $Partitions$;
11: end procedure
```

in $Ans_P^R(W)$. However, this definition has to be adapted for a non-monotonic reasoner, where there may be multiple answers for the same input. In this case, the *accuracy* is defined as follow: For each answer $ans_i \in Ans_P^{PR}(W)$, the accuracy of ans_i is computed as:

$$\max\left(\frac{|ans_i \cap ans_j|}{|ans_j|} : ans_j \in Ans_P^R(W)\right)$$

IV. Evaluation

In this section, we experimentally study the performance of the reasoner PR in the extended StreamRule framework. We demonstrate how we can: (i) speed up the non-monotonic reasoning process over increasing size of the input window, thus enhancing the scalability of the whole system and reducing the bottleneck of the one-directional process in StreamRule; (ii) increase the accuracy of the answers compared to the case of random partitioning. In the following, we describe the experimental set up, inspired by the example in Section II-A.

Rule set. We conduct experiments with two logic programs P and P' (as in Section II-A and Section II-B respectively): the first is used to investigate if the reasoner PR over P reduces the reasoning latency and improves the accuracy of the answers; the second is used to study if the reasoner PR over P' contributes to the reasoning latency and the accuracy since $G_{P'}^{inpre(P')}$ is connected and data items of some common predicates need to be duplicated in the partitioning process.

Input window. The experimental data is in RDF triple format $< s, p, o >$. We build the synthetic data by randomly generating triples where each p belongs to $inpre(P)$. For s or o, we randomly generate their values as numbers bound by n, where n is the size of the input window. We execute the reasoners R and PR over programs P and P' with increasing

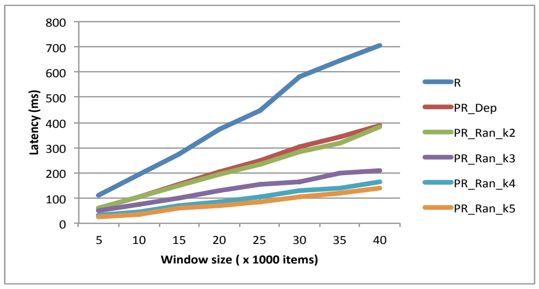

Fig. 7: Reasoning latency (program P)

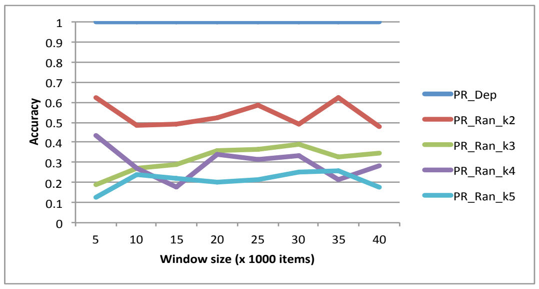

Fig. 8: Accuracy (program P)

input window size from 5000 to 40000 items. We use the ASP solver Clingo 4.3.0 [9] and Java 8. The experiment is conducted on a machine running Debian GNU/Linux 6.0.10, containing 8-cores of 2.13 GHz processor and 64 GB RAM.

Experiment with program P**.** We evaluate the performance of the reasoner PR on P with several sizes of W. We use two ways of partitioning in PR: randomly and based on dependency analysis. The input dependency of $inpre(P)$ with respect to P (as in Example 3) allows to partition W in two. For random partitioning, we split W into k = 2, 3, 4, and 5 partitions. We compare their performance with the performance of processing the entire input window W. Results of reasoning latency and accuracy are shown in Figure 7 and Figure 8 respectively, where we show that using dependency analysis substantially reduces around 50% of the latency while the accuracy is maintained (note that the number of the answer set in this case is 1). Random partitioning makes the reasoner run faster, but the accuracy decreases sharply. It is possible to observe that the reasoning time of PR in both types of partitioning is almost the same for $k = 2$ while the accuracy of the answers in our proposed partitioning method is significantly higher than in the case of random partitioning.

Experiment with program P'**.** In this experiment we used R and PR on P', with with the same input ($inpre(P') = inpre(P)$). We aim at investigating how the input dependency graph contributes to the time required for the partitioning process in particular and the whole reasoning process in general. The input dependency graph of P' with respect to $inpre(P')$ does not naturally induce subdivisions of $inpre(P')$. Our solution decomposes the input in two partitions, with duplicated predicate *car_number*. Results of this experiment are illustrated in Figure 9 (for the reasoning latency) and in Figure 10 (for the accuracy of the answers). The difference in the latency of the reasoner PR between partitioning randomly and partitioning with dependency analysis is noticeable. Time required for processing the duplicated predicate increases latency up to 30%. Note that the average percentage of instances of the duplicated predicate in a window is 25%. The accuracy of the answers remains the same as that for P.

V. Related Works

The parallel strategies were important features of database technology in the nineties in order to speeding up the execution of complex queries [5]. In Semantic Web, the parallelism in reasoning has been studied in [17], [21], [19], [20] where a set of machines is assigned a partition of the parallel computation. [17] has a distributed process over large amounts of RDF data using a proposed divide-conquer-swap strategy, which extends the traditional approach of divide-and-conquer with an iterative procedure whose result converges towards completeness over time. Similarly, [21] proposes a technique for materialising the closure of an RDF graph based on MapReduce [7]. The authors in [20] also use MapReduce to explore the reasoning in the form of defeasible logic. They restrict this logic to the argument defeasible logic. Afterwards, they apply a similar approach to systems of well-founded semantics. While the works in [17], [21] focus on monotonic reasoning, [21] and [20] examine non-monotonic reasoning over massive data. However, these attempts do not consider the streaming setting and to not rely on the stable model semantics.

In ASP, several works about parallel techniques for the evaluation of a logic program have been proposed [13], [11], [3], [6], [18], focusing on both two distinct phases of ASP computation, namely grounding and solving. Concerning the parallelisation of the grounding phase, the work in [3] is applicable only to a subset of the program rules. Therefore, in general, this work is unable to exploit parallelism fruitfully in the case of programs with a small number of rules. [6] explores some structural properties of the input program via the defined dependency graph in order to detect subprograms that can be evaluated in parallel. [18] extends this work with parallelism in three different steps of the grounding process: components, rules, and single rule level. The first level divides the input program into subprograms, according to the dependency graph among IDB predicates of that program. The second level allows for concurrently evaluating the rules within each subprogram. The third level partitions the extension of a single rule literal into a number of subsets. This step is especially efficient when the input program consists of few rules and two first levels have no effects on the evaluation of the program. For the solving step which is carried out after the grounding step, [13] proposes a generic approach to distribute the searching space in order to find the answer sets, which permits exploitation of the increasing availability of clustered and/or multiprocessor machines. [11] introduces a conflict-driven algorithm to compute the answer sets based on constraint processing and satisfiability checking. In short, [3], [6], [18] focus on parallel instantiation by splitting a logic program in order to obtain a smaller ground program, [13], [11] parallel compute the answer sets from that ground program. These approaches have been implemented in state-of-the-art ASP solver such as Clingo and DLV. In this paper, we are not partitioning the logic program. We are focusing on partitioning the input and evaluating each partition on a different copy of the whole program with the intuition that this approach is data-

[9]http://sourceforge.net/projects/potassco/files/clingo/4.3.0/

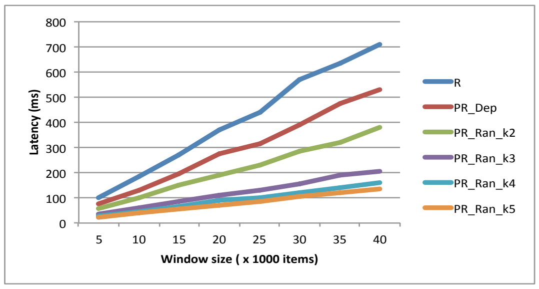

Fig. 9: Reasoning latency (program P')

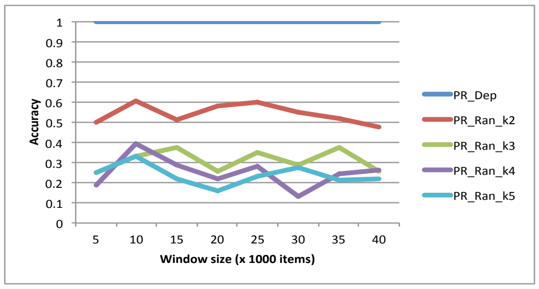

Fig. 10: Accuracy (program P')

driven and can result in a much faster run-time analysis since it does not have to consider the whole program but only the part of it that is interesting based on the streaming input.

VI. Conclusions and Future Work

In this work, we have studied the problem of enhancing the scalability of a declarative stream reasoning system that relies on the combination of semantic query processing and disjunctive logic reasoning. We have analysed the input dependency graph to model the relations among predicates of data items. We have elicited our approach as an extension of the ASP-based reasoning layer of the StreamRule framework, in the data partitioning process. The data partitioning process controlled by dependencies we have proposed can: (i) reduce the latency of the reasoning layer, and (i) increase the accuracy of the answers. In the future, more experiments on different rule sets and data will have to be considered for a more extensive evaluation. Moreover, we believe that due to the definition of the input dependency graph, the accuracy of the answers can be guaranteed. Therefore, providing a proof of correctness of answers is also in our next step.

An interesting further extension lies in the input dependency at the atom level. In our current solution, we are considering input dependencies among predicates to split the window. However, we have observed input dependency at the atom level in form of self-loops in our graph. We believe that dependencies among ground atoms have an important effect on computation. This requires studying the distribution characteristics of atoms by looking closer at the rule set. More generally, we will extend this approach to consider both predicate and atom levels.

Acknowledgment

This research is partially supported by Science Foundation Ireland (SFI) grant No. SFI/12/RC/2289 and Insight-Cisco Systems Galway targeted project grant No. RIB1144.

References

[1] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[2] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in etalis. *Semantic Web*, 3(4):397–407, 2012.

[3] M. Balduccini, E. Pontelli, O. Elkhatib, and H. Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[5] F. Cacace, S. Ceri, and M. Houtsma. A survey of parallel execution strategies for transitive closure and logic programs. *Distributed and Parallel Databases*, 1(4):337–382, 1993.

[6] F. Calimeri, S. Perri, and F. Ricca. Experimenting with parallelism for the instantiation of asp programs. *Journal of Algorithms*, 63(1):34–54, 2008.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] T. M. Do, S. W. Loke, and F. Liu. Answer set programming for stream reasoning. In *Advances in Artificial Intelligence*, pages 104–109. Springer, 2011.

[9] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. *Dlv-hex: Dealing with semantic web under answer-set programming*. In: Proc. of ISWC, 2005.

[10] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Answer set programming for stream reasoning. *CoRR, abs/1301.1392*, 2013.

[11] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[12] S. Germano, T.-L. Pham, and A. Mileo. Web stream reasoning in practice: on the expressivity vs. scalability tradeoff. In *Web Reasoning and Rule Systems*, pages 105–112. Springer, 2015.

[13] J. Gressmann, T. Janhunen, R. E. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A platform for distributed answer set solving. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 227–239. Springer, 2005.

[14] R. Lambiotte, J.-C. Delvenne, and M. Barahona. Laplacian dynamics and multiscale modular structure in networks. *arXiv preprint arXiv:0812.1770*, 2008.

[15] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer, 2011.

[16] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. Streamrule: a nonmonotonic stream reasoning system for the semantic web. In *Web Reasoning and Rule Systems*, pages 247–252. Springer, 2013.

[17] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen. Marvin: Distributed reasoning over large-scale semantic web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305–316, 2009.

[18] S. Perri, F. Ricca, and M. Sirianni. Parallel instantiation of asp programs: techniques and experiments. *Theory and Practice of Logic Programming*, 13(2):253–278, 2013.

[19] I. Tachmazidis, G. Antoniou, and W. Faber. Efficient computation of the well-founded semantics over big data. *arXiv preprint arXiv:1405.2590*, 2014.

[20] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas. Towards parallel nonmonotonic reasoning with billions of facts. In *KR*, 2012.

[21] J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen. Scalable distributed reasoning using mapreduce. In *Proceedings of the 8th International Semantic Web Conference*, pages 634–649. Springer-Verlag, 2009.