Semantic Web 1 (0) 1–5
IOS Press

Enhancing the Scalability of Expressive Stream Reasoning via input-driven Parallelization

Thu-Le Pham [a,*], Muhammad Intizar Ali [a] and Alessandra Mileo [b]
[a] *Insight Centre for Data Analytics, National University of Ireland, Galway, IDA Bussiness Park, Lower Dangan, Galway, Ireland*
E-mails: thule.pham@insight-centre.org, ali.intizar@insight-centre.org
[b] *Insight Centre for Data Analytics, Dublin City University, Glasnevin, Dublin 9, Dublin, Ireland*
E-mail: alessandra.mileo@insight-centre.org

Editors: Daniele Dell'Aglio, University of Zurich, Switzerland; Thomas Eiter, TU Vienna, Austria; Fredrik Heintz, Linköping University, Sweden; Danh Le Phuoc, TU Berlin, Germany
Solicited reviews: David Bowden, Dell EMC Research Europe, Cork, Ireland; Konstantin Schekotihin, Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria; Two anonymous reviewers

Abstract. Stream reasoning is an emerging research area focused on providing continuous reasoning solutions for data streams. The exponential growth in the availability of streaming data on the Web has seriously hindered the applicability of state-of-the-art expressive reasoners, limiting their applicability to process streaming information in a scalable way. In this scenario, in order to reduce the amount of data to reason upon at each iteration, we can leverage advances in continuous query processing over Semantic Web streams. Following this principle, in previous work we have combined semantic query processing and non-monotonic reasoning over data streams in the StreamRule system. In the approach, we specifically focused on the scalability of a rule layer based on a fragment of Answer Set Programming (ASP). We recently expanded on this approach by designing an algorithm to analyze input dependency so as to enable parallel execution and combine the results. In this paper, we expand on this solution by providing i) a proof of correctness for the approach, ii) an extensive experimental evaluation for different levels of complexity of the input program, and iii) a clear characterization of all the algorithms involved in generating and splitting the graph and identifying heuristics for node duplication, as well as partitioning the reasoning process via input splitting and combining the results.

Keywords: Semantic Web, stream reasoning, non-monotonic reasoning, Answer Set Programming, parallel reasoning, data partitioning, dependency graph

1. Introduction

The variety of real-world applications in several domains, such as the Internet of Things, Social Networks and Smart Cities, requires reasoning capabilities that can handle incomplete and potentially inconsistent input streams, and extract knowledge from them to support decision making. While semantic technologies for handling data streams focus on query pattern matching and have limited support for complex reasoning capabilities, logic-based non-monotonic reasoning approaches are very expressive but can be quite costly in terms of efficiency. Expressive stream reasoning for the Semantic Web explores advances in semantic stream processing technologies for representing and processing data streams on the one hand, and non-monotonic reasoning approaches for performing com-

*Corresponding author. E-mail: thule.pham@insight-centre.org.

plex rule-based inference on the other hand. This combination is based on the principle of having a 2-tier approach where: i) a semantic stream query processor is used to filter semantic data elements (typically RDF triples), and ii) a non-monotonic reasoner is used for computationally intensive tasks over the filtered data. Since the grounding phase in rule-based inference is responsible for the size of the program to be evaluated, such a combined approach improves the scalability of complex reasoning over Semantic Web streams by reducing the input to the non-monotonic reasoner.

Current expressive reasoning systems over RDF data streams, like ASR [12], EP-SPARQL [2], and StreamRule [24], support non-monotonic reasoning over data streams in different ways. In particular, ASR uses the DLVhex solver [14], EP-SPARQL uses ETALIS [3] which is implemented based on SWI-Prolog[1], and StreamRule uses the Clingo solver [16] as a subprocess to infer new knowledge from data streams and a given rule set. SWI-Prolog is a Prolog engine which is built on SLD-resolution and unification as the basic mechanism to manipulate data structures while DLVhex and Clingo are ASP systems which are based on the stable model (answer set) semantics of logic programming [13]. Considering the expressive power of ASP and higher declarativity compared to Prolog, we focus on ASP-based reasoning.

In order to support ASP solvers for reasoning about RDF data streams, a middle layer is required for transformation between data formats. For example, the StreamRule system intercepts the query results (output RDF stream) filtered by the RDF Stream Processing (RSP) engine and translates them into ASP syntax before streaming them into the ASP reasoner Clingo. Given the data transformation overhead, the 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 the new input window arrives, in order to ensure the stability of the whole system. This requires optimization techniques that can further speed up the processing [19].

We address this scalability issue by an approach to parallelization based on splitting the input stream (not the logic program) that we have first introduced in [27]. We extend our preliminary work from [27] in this paper with the following key contributions:

- we propose a better characterization of our formal algorithm for analyzing dependencies among input data based on the structure of a given logic program (a set of logic rules). This program is constructed under the stratified negation fragment of normal ASP [13], which ensures uniqueness of the solution; the algorithm characterizes different relationships between two predicates appearing in the input data in form of so-called input dependency graph;
- we provide a process that uses this input dependency graph to construct a plan for partitioning input data; when the graph is connected, it is decomposed into subgraphs such that the number of common nodes is as small as possible; this partitioning plan will guide the reasoning process to split input data on-the-fly;
- we fully implement our approach as an extension of StreamRule for validation and testing of our algorithms. With StreamRule, our reasoner does not need to deal with input data elements that are unrelated to the reasoning task since they are filtered out by the stream processor. We believe this idea of filtering massive input to related input for specific complex reasoning tasks is promising for handling scalability of stream reasoning over Semantic Web streams.
- we provide a formal proof that the correctness for the approach under the stable model semantics of ASP is guaranteed;
- we conduct a detailed experimental evaluation on the effectiveness of our approach via experiments with different levels of expressivity of the logic program, namely: positive rules, recursive positive rules, and stratified negation. Results show that our approach can achieve higher expressivity and higher scalability compared to state-of-the-art stream processing engines.

The remainder of this paper is organized as follows. Section 2 provides the necessary preliminaries on ASP, the StreamRule idea and conceptual framework, and introduces our motivating example. Section 3 defines in details our input dependency analysis process, including the generation of the graph, the heuristics for node duplication and the process of building a partitioning plan. In Section 4, we report on the extension of the StreamRule system with components in charge of partitioning and combining the results of the inference process, and we provide a proof of correctness of the results for the proposed method. Section 5 pro-

[1] http://www.swi-prolog.org

vides an extensive evaluation of our approach through three different experiments. A comprehensive discussion of related work is given in Section 6, followed by concluding remarks and directions for future work in Section 7.

2. Preliminaries & Motivating Example

2.1. Answer set programming

Answer Set Programming (ASP) is a declarative problem solving paradigm with a rich yet simple modeling language and high performance solving capabilities for computationally hard problems. ASP is rooted in deductive databases, logic programming and constraint solving [13]. For this paper, we focus on normal ASP with stratified negation.

Syntax.

In ASP, a variable or a constant is a *term*[2]. An *atom* is $p(t_1, ..., t_n)$, where p is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal not* p, where p is an atom. A *normal logic program* is the program that consists of rules of the form:

$$q \leftarrow p_1, ..., p_k, not\ p_{k+1}, ..., not\ p_m$$

where $q, p_1, ..., p_m$ are atoms and $m \geqslant k \geqslant 0$.

Given a rule r as above, we define $head(r) = \{q\}$ as the head of r, while $body(r) = \{p_1, ..., p_k, not\ p_{k+1}, ..., not\ p_m\}$ is the body of r. $body^+(r)$ (respectively, $body^-(r)$) denotes the set of atoms occurring positively (respectively, negatively) in $body(r)$. A rule where $head(r) = \emptyset$ is referred to as an *integrity constraint*. A rule where $body(r) = \emptyset$ is called a *fact*. A term, an atom, a literal, a rule, a program is *ground* if no variable appears in it. According to the database terminology, a predicate occurring only in *facts* is referred to as an *EDB (extensional database)* predicate, all others as *IDB (intensional database)* predicates. EDB predicates are relations stored in a database, while IDB ones are relations defined by one or more rules. Thus, an IDB predicate can appear in the body or head of a rule while an EDB predicate is only in the body. We only allow *stratified negation* to appear in a program, i.e., the program should contain no recursion through negation. Intuitively, recursion through negation (or unstratified negation) happens when two or more predicates are mutually defined over *not* such as $\{b \leftarrow not\ a, a \leftarrow not\ b\}$.

Semantics.

Let P be a program. The *Herbrand Universe*, U_P, of P is a set of all constants appearing in P. The *Herbrand Base*, B_P, of P is a set of all ground atoms constructible from the predicate symbols appearing in P and the constants of U_P. $ground(P)$ denotes the set of all the ground instances of the rules occurring in P. An *interpretation*, M, for P is a subset of B_P. A ground rule $r_g \in ground(P)$ is satisfied with respect to M if $body^+(r_g) \subseteq M$ and $body^-(r_g) \cap M = \emptyset$ only if $head(r_g) \cap M \neq \emptyset$. M is a *model* of P if M satisfies all ground rules in $ground(P)$. The *reduct*, P^M, of P relative to M is given by $\{head(r_g) \leftarrow body^+(r_g) | r_g \in ground(P)$ and $body^-(r_g) \cap M = \emptyset\}$. M is an *answer set* of P if it is a minimal model of P^M (i.e., M is a model of P^M and $\nexists M' \subset M$ such that M' is a model of P^M). If P is stratified then M is a unique answer set of P.

2.2. StreamRule

StreamRule is a framework that combines the latest advances in stream query processing for Semantic Web data, with non-monotonic stream reasoning. The approach is based on the assumption that not all raw data from the input stream might be relevant for the complex reasoning, and the stream query processing can help to reduce the information load over the logic-based stream reasoner. The conceptual architecture of *StreamRule* is shown in Figure 1. Abstraction and filtering on raw streaming data are performed by a *stream query processor* using query patterns as filters. The filtered stream is processed by a *data format processor* and returned as input facts to a *non-monotonic rule engine* together with the declarative encoding of the problem at hand. The output of the rule engine, which we call solutions or *answer sets*, is fed into the *data format processor* for transformation to any other format (such as back to RDF triples) for further processing.

The main limitation of StreamRule is that the stability of the system depends on the ability of the reasoner to produce results faster than the next input window arrives. For this reason, as a first step in targeting the scalability challenge, we focused on a mechanism to enhance the processing time of the logic-based rea-

[2]We do not consider functional symbols, although they are currently allowed in some extensions of ASP.

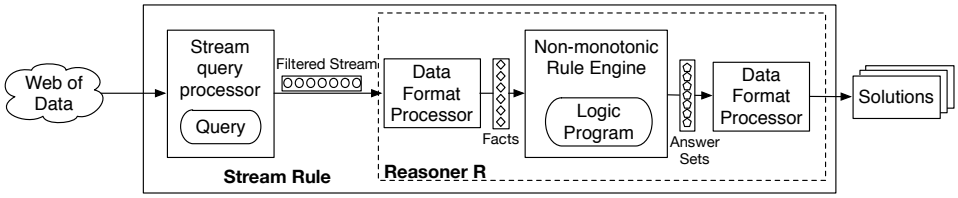

Fig. 1. Conceptual architecture of StreamRule

soner by designing a formal strategy for input dependency analysis, and using it to enable parallelism at the reasoning layer of StreamRule (the reasoner R in Figure 1). A follow-up of the proposed approach is that we can gather information on the process at the reasoning layer that can potentially be used to dynamically adapt the parameters of the RSP engine for adaptive scalability management. We do not tackle this aspect in this paper but it is part of our ongoing work as discussed in Section 7.

For the rest of the paper, we use *RSP engine* to refer to the semantic stream query processing engine (e.g., C-SPARQL [5]), *solver* to refer to the *non-monotonic rule engine* (e.g., Clingo), *reasoner R* to refer to the subprocess in StreamRule which includes the *solver* and the *data format processor* (the dashed box in Figure 1), and *reasoner PR* (the gray box in Figure 6) to refer to the optimized version of R with the parallel approach that will be detailed in the following sections. Before introducing our motivating example, we also want to briefly introduce some notation that will be used therein and after. In the reasoner R, a given *logic program* (or *program*), denoted as P, is a set of rules (with stratified negation) in ASP. $pre(P)$ denotes the set of predicates in P. $inpre(P)$ denotes predicates provided as input data items of P. As illustrated in Figure 1, the reasoner R receives input data items from the RSP engine. We assume that unrelated predicates are filtered out by the RSP engine through appropriate queries. In this way, $inpre(P) \subseteq pre(P)$. 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 referred to as *ground atoms*. $pre(W)$ defines the set of predicates of ground atoms in W. Therefore, $pre(W) \subseteq inpre(P)$.

2.3. Motivating Example

Consider the following example: A city manager wants to know real-time events happening in the city in order to make informed decisions on traffic management, reaction to vandalism/crime, management of traffic congestions, reduction of risks for drivers/cyclists/pedestrians, and so on. To do that, he deploys an instance of the StreamRule system that integrates and filters relevant semantic streams from different sources (via RSP engine queries) and uses them to detect events of interest, such as *traffic_jam* and *car_fire* as defined in the logic program P in Listing 1. P is given as input to the solver in 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 RSP engine.

As an illustrative example, assume at time t, a filtered input window (in ASP format) arrives as follows: W = {*average_speed(newcastleRoad, 10)*, *car_number(newcastleRoad, 55)*, *traffic_light(newcastleRoad)*, *car_in_smoke(car1, high)*, *car_speed(car1, 0)*, *car_location(car1, danganRoad)*}. This example is probably not presenting issues in terms of performance, but as the number of cars, segments, traffic lights and other events increases, the scalability of the system becomes an issue.

In order to process W faster, partitioning W randomly as in [19] could generate wrong results. For example W_1 = {*average_speed(newcastleRoad, 10), car_number(newcastleRoad, 55), car_in_smoke(car1, high)*} and W_2 = {*traffic_light(newcastleRoad), car_speed(car1, 0), car_location(car1, danganRoad)*}. Reasoning in parallel over these two input partitions produces as a result the event *traffic_jam(newcastleRoad)* and the action *give_notification(newcastleRoad)* is triggered, which is not correct. The accurate answer is the event *car_fire(danganRoad)* detected and the notification about the *danganRoad* segment. Partitioning randomly the input stream may reduce the processing time of a logic-based reasoner but we may lose the accuracy of the results in return. Therefore, the partitioning process should consider the relations between ground atoms in the input window, and distribute the computation accordingly across multiple instances of

```
(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: Sample rules for detecting events

the rule set (logic program). Note that this approach is different from distributing the processing by splitting the rules, and it targets instead the input predicates. How this input analysis is done will be detailed in the following section.

3. Input Dependency Analysis

In this section, we discuss the problem of analyzing the dependency of input elements in a window W for the reasoner R with respect to a set of ASP rules in a program P with stratified negation. We first introduce the concept of input dependency graph that shows how input data items in W relate to each other with respect to the logic program P (Section 3.1). Thereafter, we present a heuristic-based algorithm for creating a partitioning plan which is used to split streaming input data on the fly (Section 3.2).

3.1. Input Dependency Graph

In order to build an input dependency graph among data items in an input window W, we follow a 2-step approach: first, the *dependency graph* as defined in [10] is extended to capture additional relationships that go beyond dependencies among IDB predicates and also consider EDB predicates; second, only predicates appearing in W and their dependencies will be extracted from the graph built in the first step, in order to capture only the relationships among input data.

The concept of dependency graph has been widely used in ASP as a tool to analyze the structure of non-ground answer set programs [10, 26]. It has been efficiently used in parallel instantiation algorithms that generate a much smaller ground program equivalent to a given logic program. Note that 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. The instantiation process in ASP can be expensive from a computational viewpoint and the size of the ground program has a huge effect on the performance of the solver. To address this issue, the idea of parallel grounding has been investigated, which relies on the concept of dependency graph. As defined in [10], a dependency graph G is a directed graph where nodes are IDB 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 grounded in parallel.

However, in this paper, we are not partitioning the logic program for the grounding process. We are focusing instead on partitioning the input on-the-fly and evaluating each partition in parallel with a copy of the whole program P. Our approach intuitively generates a smaller ground program (because of the reduced input) and a smaller search space, speeding up both grounding and solving. It is to be noted that this is not the reason why we restrict the approach to consider programs with stratified negation: we restrict the expressivity of ASP programs to ensure the correctness of results of the parallel reasoning. This is not guaranteed if we have unstratified negation.

The reasons for us to follow the input partitioning approach are: (i) input data (or input facts) have a significant impact on the reasoning performance in a streaming scenario and can affect results more than the complexity of the rules, and (ii) in the context of dynamic environments, the amount of input data at each execution varies in terms of rate and size, thus having different effects on performance. We assume that the input predicates can be either IDB or EDB 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.

In order to capture this aspect, we first define an *extended dependency graph* from the definition in [10]. 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.

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

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 r in P. Moreover, $(p_u, p_u) \in E_{P_1}$ if $p_u \in body^-(r)$.

(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.

Note that p_u, q_u, p_d, q_d are predicates that can appear in 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.*

Based on the extended dependency graph, we introduce the concept of *input dependency graph* of P with respect to $inpre(P)$. This input dependency graph describes how predicates in $inpre(P)$ depend on each other. Below, we describe the meaning of *direct path* that is used to build the input dependency graph.

Definition 2. *Given the extended dependency graph $G_P = \langle N_P, E_P \rangle$ of the logic program P, a directed path from node p_1 to node p_n is a sequence of nodes $p_1, p_2, ..., p_n$ such that $p_i \in N_P, i = 1..n$ and $\langle p_j, p_{j+1} \rangle \in E_{P_2}, j = 1..n-1$.*

Definition 3. *Let P be a logic program, $G_P = \langle N_P, E_P \rangle$ be an extended dependency graph of P, 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) $p \neq q$ and 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: one is from p_1 to p_i if $p_1 \neq p_i$ and the other is from p_n to p_{i+1} if $p_n \neq p_{i+1}$.

ii) $p = q$ and $((p, p) \in E_{P_1}$ or $\exists u \in N_P, (u, u) \in E_{P_1}, \langle p, u \rangle \in E_{P_2})$.

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 4. *Let P be a logic program and $inpre(P)$ be a set of input predicates of P. 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 3, the first condition represents dependencies among all ground atoms of two different predicates in $inpre(P)$ (predicate level) while the second condition shows dependencies among ground atoms of a self-loop predicate (atom level). Note that a self-loop predicate is one that has an edge connecting the predicate to itself. When two different predicates depend on each other or a predicate depends on itself, it means that their ground atoms can contribute to infer a new fact by firing a single rule or multiple rules. Therefore, all ground atoms of dependent predicates need to be processed together in order to guarantee that rules in P are fired properly and to ensure correctness of results.

We will conclude this section by reporting the two algorithms that generate an input dependency graph with a given extended dependency graph and a set of input predicates. The algorithm for building an extended dependency graph is not reported because it is trivial from Definition 1.

Algorithm 1 creates an input dependency graph as defined in Definition 3. $N_P^{inpre(P)}$ and $E_P^{inpre(P)}$ contain vertexes and edges of the graph. At the beginning, each predicate in $inpre(P)$ is identified as a vertex (Line 2). Each vertex is checked to see if it depends on other vertexes according to the conditions in Definition 3. In Line 5-9, the algorithm checks condition (i) in Definition 3 by calling the underlying function *CheckDependency* which is detailed in Algorithm 2. Line 10-17 create a self-loop for a vertex if condition (ii) in Definition 3 holds. First, it takes a self-loop in E_{P_1} that is related to the current vertex (Line 10-12). Then, it creates a self-loop for a vertex if this vertex implies another self-loop vertex (Line 13-17).

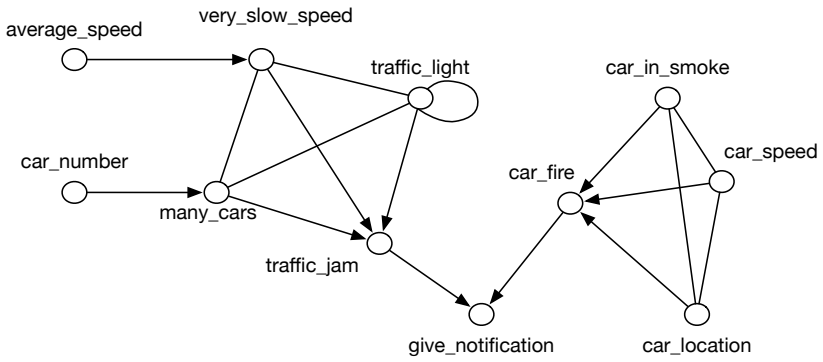

Fig. 2. Extended dependency graph G_P

The goal of the function *CheckDependency* is to check if two separated vertexes v_1 and v_2 depend on each other as per condition (i) in Definition 3. There is a basic dependency between two predicates if there is an undirected link between them (Line 12-13). Otherwise, the algorithm will find if there are two direct paths connected by an undirected edge between those two vertexes. This function is extended from the breadth-first search algorithm to discover those paths. This algorithm will terminate at Line 13 or when all vertexes are checked.

3.2. Partitioning Plan

In this section, we show how to use the input dependency graph for building a plan to partition streaming data on the fly. The input dependency graph is defined as an undirected graph. Therefore, we consider separately two cases based on the connectivity of the graph: not connected and connected[3].

The input dependency graph $G_P^{inpre(P)}$ that is not connected induces naturally a subdivision of the graph into several *connected components* (or *components*). A connected component of an undirected graph is a maximal connected subgraph of the graph. For instance, $G_P^{inpre(P)}$ in Figure 3 is decomposed into two components which have separate sets of nodes from $inpre(P)$: {*average_speed*, *traffic_light*, *car_number*} and {*car_in_smoke*, *car_speed*, *car_location*}. These sets of nodes are used as a partitioning plan in the partitioning process for splitting ground atoms in a window on-the-fly.

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

Algorithm 1 Creating input dependency graph

Input: an extended dependency graph G_P and a set of input predicates $inpre(P)$
Output: an input dependency graph $G_P^{inpre(P)}$

1: **procedure** IDG(G_P, $inpre(P)$)
2: $N_P^{inpre(P)} \leftarrow inpre(P)$
3: $E_P^{inpre(P)} \leftarrow \{\}$
4: **for** $v_1 \in N_P^{inpre(P)}$ **do**
5: **for** $v_2 \in N_P^{inpre(P)}$ **do**
6: **if** $CheckDependency(v_1, v_2, G_P)$ **then**
7: $E_P^{inpre(P)} = E_P^{inpre(P)} \cup \{(v_1, v_2)\}$
8: **end if**
9: **end for**
10: **if** $(v_1, v_1) \in E_{P_1}$ **then**
11: $E_P^{inpre(P)} = E_P^{inpre(P)} \cup \{(v_1, v_1)\}$
12: **end if**
13: **for** $v \in N_P$ **do**
14: **if** $(v, v) \in E_{P_1}$ & $\langle v_1, v \rangle \in E_{P_2}$ **then**
15: $E_P^{inpre(P)} = E_P^{inpre(P)} \cup \{(v_1, v_1)\}$
16: **end if**
17: **end for**
18: **end for**
19: **return** $G_P^{inpre(P)} = \langle N_P^{inpre(P)}, E_P^{inpre(P)} \rangle$
20: **end procedure**

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

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

Algorithm 2 Check dependency between 2 vertexes

Input: two vertexes v_1, v_2 and an extended dependency graph G_P
Output: true/false

```
1:  procedure CHECKDEPENDENCY(v1, v2, G_P)
2:      queueV1 ← [v1]
3:      queueV2 ← [v2]
4:      checked ← {}
5:      while queueV2 ≠ ∅ do
6:          tempV2 ← queueV2.remove(0)
7:          while queueV1 ≠ ∅ do
8:              tempV1 ← queueV1.remove(0)
9:              if (tempV1, tempV2) ∈ checked then
10:                 continue
11:             end if
12:             if (tempV1, tempV2) ∈ E_P1 then
13:                 return true
14:             else
15:                 for ⟨tempV1, cV⟩ ∈ E_P2 do
16:                     Add cV into queueV1
17:                 end for
18:             end if
19:             Add (tempV1, tempV2) into checked
20:         end while
21:         Add v1 into queueV1
22:         for ⟨tempV2, cV⟩ ∈ E_P2 do
23:             Add cV into queueV2
24:         end for
25:     end while
26:     return false
27: end procedure
```

Assume that $inpre(P') = inpre(P)$. The input dependency graph $G_{P'}^{inpre(P')}$ is shown in Figure 4. This graph is connected. Our data partitioning approach cannot be applied if the input dependency graph cannot be decomposed as in this case. To cope with this issue, we introduce the *decomposing process* to divide the graph by duplicating some common nodes. Algorithm 3 describes this process. The algorithm has two main steps: (1) finding all maximal cliques of the graph, (2) (heuristic-driven) merging of two cliques for which the ratio between common vertexes and different vertexes is bigger than 0.5 (Line 8). A clique C is a subset of the node set of a graph, such that there exists an edge between each pair of nodes in C. A maximal clique is a clique that cannot be extended by adding more nodes. Line 2 computes all maximal cliques of the input dependency graph by using a function supported in the *Toolkit* class of the *graphstream* package [4]. After that, the algorithm checks for each pair of cliques whether they can be merged. This algorithm always terminates when it can not find any pair of cliques that verify the condition in Line 8.

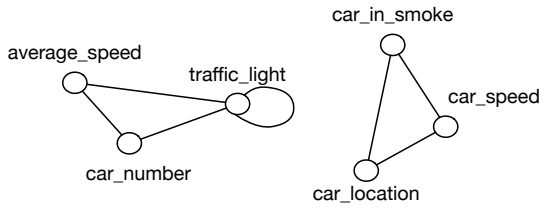

Fig. 3. Input dependency graph $G_P^{inpre(P)}$

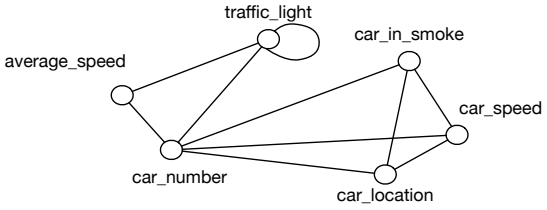

Fig. 4. Input dependency graph $G_{P'}^{inpre(P')}$

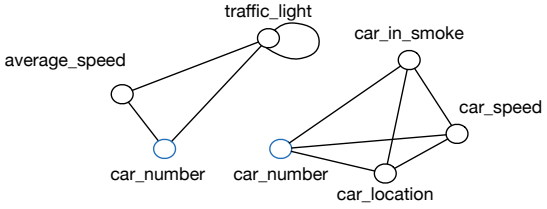

Fig. 5. Output of the decomposing process for $G_{P'}^{inpre(P')}$

Example 3. *Consider the input dependency graph $G_{P'}^{inpre(P')}$ in Figure 4. Step 1 of the Algorithm 3 finds two maximal cliques C_1 = {traffic_light, average_speed, car_number} and C_2 = {car_number, car_in_smoke, car_speed, car_location}. These two cliques are not merged since the rate between common predicates and different predicates is $\frac{1}{5} < 0.5$. Therefore, they are considered as two sets of nodes in the partitioning plan (see Figure 5), which guides the parallel reasoning process.*

[4] http://graphstream-project.org

Algorithm 3 Decomposing process

Input: input dependency graph $G_P^{inpre(P)}$
Output: Partitioning plan

```
1:  procedure DECOMPOSEIDG(G_P^{inpre(P)})
2:      cliques ← getMaximalCliques(G_P^{inpre(P)})
3:      while true do
4:          flag ← false
5:          for each (C_1 ≠ C_2) ∈ cliques do
6:              nCNodes ← |intersect(C_1, C_2)|
7:              nDNodes ← |C_1| + |C_2| - 2*nCNodes
8:              if nCNodes/nDNodes > 0.5 then
9:                  Add merge(C_1, C_2) into cliques
10:                 Remove C_1, C_2 from cliques
11:                 flag ← true;
12:                 break
13:             end if
14:         end for
15:         if !flag then
16:             break
17:         end if
18:     end while
19:     return cliques
20: end procedure
```

4. Parallel Reasoning in StreamRule

4.1. Implementation

The StreamRule framework extended with the partitioning process described in this paper is shown in Figure 6. The extension consists of the *partitioning handler* and the *combining handler* in the reasoning layer. The partitioning handler splits an input window W coming from the RSP engine into several sub-windows taking into account the input dependency. The combining handler combines outputs from parallel instances of the reasoner. For the realization of the partitioning process, the analysis of input dependency is made available within the framework initially 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 3. Then the graph decomposing process described in Section 3.2 builds a partitioning plan by decomposing this graph into several components, with duplicated predicates when needed.

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 4 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 groups belong to based on the partitioning plan (Line 5). Finally, it adds that group into the proper partitions corresponding to those IDs.

The combining handler. Given a program P under stratified negation and an input window W, the answers provided by R over P and W (notated as $Ans_P(W)$) are computed as:

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

Where W_i ($i = 1..n$) are partitions of W provided by the partitioning handler.

Algorithm 4 Partitioning method

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

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

4.2. Correctness

In order to ensure our approach provides all and only the expected results when the input is split and processed in parallel, in this section we provide a sketch of the correctness proof.

Proposition 1. *Given $G_P^{inpre(P)}$ that is not connected, $G_1, ..., G_n$ ($n > 1$) are connected components of $G_P^{inpre(P)}$, and W is an input window such that $pre(W) \subseteq inpre(P)$:*

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

where $W = \bigcup_{i=1}^{n} W_i$, and $pre(W_i)$ is the set of nodes of G_i.

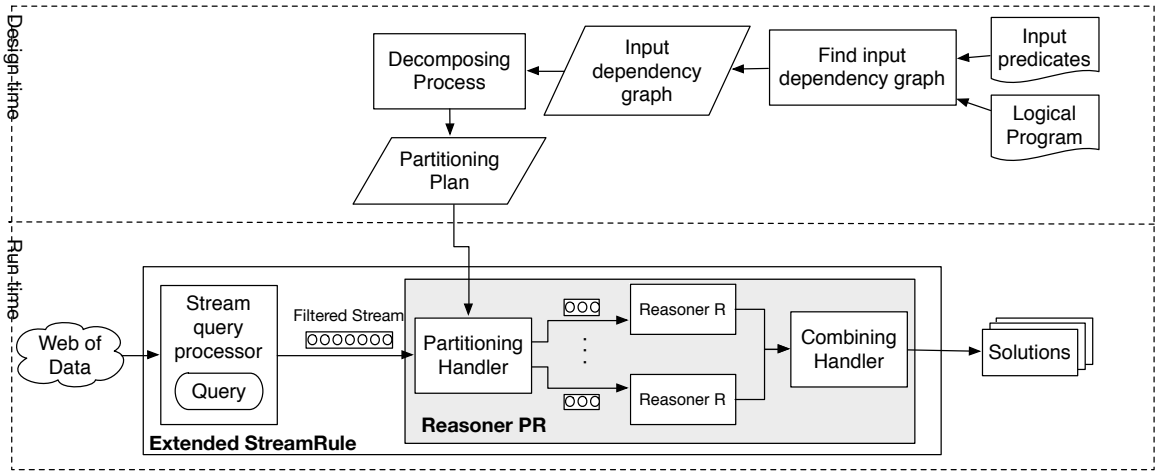

Fig. 6. The Extended StreamRule

Proof. We introduce some notations that are used in the proof:

- $pre(body(r))$: a set of predicates appearing in the body of rule r.
- $pre\_head(r)$: a predicate appearing in the head of rule r.
- $ground(p)$: a set of ground atoms over the predicate p.

Suppose $a \in Ans_P(W)$, we consider the following cases:

- a is created by firing one rule r in P
  $\Rightarrow \forall_{p_i,p_j \in pre(body(r))}(p_i \neq p_j), (p_i, p_j) \in E_P^{inpre(P)}$
  $\Rightarrow \exists i \in [1,n] : \forall p \in pre(body(r)), ground(p) \subset W_i$
  $\Rightarrow a \in Ans_P(W_i)$
  $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$
- a is created by firing two rules r_1, r_2 in P
  $\Rightarrow pre\_head(r_1) \in pre(body(r_2))$
  - If $pre(body(r_2)) = \{pre\_head(r_1)\}$
    $\Rightarrow \forall p \in pre(body(r_1)), (pre\_head(r_1), p) \notin E_P^{inpre(P)}$
    $\Rightarrow \exists W_i \neq W_j : pre(body(r_1)) \subset pre(W_i)$ and $pre\_head(r_1) \in pre(W_j)$
    $\Rightarrow \forall p \in pre(body(r_1)), ground(p) \subset W_i$ and $ground(pre\_head(r_1)) \subset W_j$
    $\Rightarrow a \in Ans_P(W_i)$ (by firing both r_1 and r_2) or $a \in Ans_P(W_j)$ (by firing r_2)
    $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$
  - Else
    $\Rightarrow \forall p \in pre(body(r_1)), \forall q \in pre(body(r_2)), (p,q) \in E_P^{inpre(P)}$
    $\Rightarrow \exists i = [1..n] : \forall p \in pre(body(r_1)) \cup pre(body(r_2)), ground(p) \subset W_i$
    $\Rightarrow a \in Ans_P(W_i)$
    $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$
- Similarly, when a is created by firing k rules $r_1, ..., r_k$ in P
  $\Rightarrow \exists i \in [1..n] : \forall p,q \in \bigcup_{j=1}^{k} pre(body(r_j)), (p,q) \in E_P^{inpre(P)} \rightarrow p,q \in pre(W_i)$
  $\Rightarrow \exists W_i : a \in Ans_P(W_i)$
  $\Rightarrow a \in \bigcup_{i=1}^{n} Ans_P(W_i)$

Suppose $a \in \bigcup_{i=1}^{n} Ans_P(W_i) \Rightarrow \exists i \in [1..n] : a \in Ans_P(W_i)$

- If a is created by firing a set of positive rules
  $\Rightarrow a \in Ans_P(W)$ because $W_i \subset W$
- If a is created by firing a set of rules (e.g., $r_1, ..., r_k$) with negation-as-failure
  $\Rightarrow \forall p \in \bigcup_{i=1}^{k} pre(body^{-}(r_j)), ground(p) \subset W_i$ and $\nexists W_j \neq W_i$: $ground(p) \subset W_j$
  $\Rightarrow a \in Ans_P(W)$.

Proposition 2. *Given $G_P^{inpre(P)}$ that is connected and W is an input window such that $pre(W) \subseteq inpre(P)$:*

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

where $W = \bigcup_{i=1}^{n} W_i$, and $pre(W_i)$ are computed by the Algorithm 3.

Proof. When $G_P^{inpre(P)}$ is connected, Algorithm 3 decomposes $inpre(P)$ into $pre(W_i), i = 1..n$ and the

intersection of any two sets $pre(W_i)$ and $pre(W_j)$ ($pre(W_i) \neq pre(W_j)$) may be not empty. Without loosing generality, assume that $pre(W_i) \cap pre(W_j) = \{p\}, p \in inpre(P)$. The partitioning process in Algorithm 4 adds all ground atoms over p into both W_i and W_j. In this way, we do not loose the dependencies between p with other predicates in $pre(W_i)$ (or in $pre(W_j)$). Therefore, the correctness of the parallel reasoning process is maintained as proved in Proposition 1.

5. Evaluation

We evaluate the performance of our optimized reasoner *PR* on input programs with different levels of expressivity: positive rules (experiment 1), positive recursive rules (experiment 2), and stratified negation rules (experiment 3). In each experiment, we compare the performance against state-of-the-art engines supporting the same level of expressivity with respect to two metrics: latency and memory consumption. Latency refers to the time consumed by the engines between input arrival and output generation while memory consumption reflects the usage of system memory during execution. The experiments were conducted on a machine with 24-core Intel(R) Xeon(R) 2.40 GHz and 96G RAM. We used Java 1.8 with heap size from 5GB to 20GB for C-SPARQL and Clingo 4.5.4 for the reasoners. The experiments code and data is available at https://github.com/ThuLePham/SR_Experiments. The empirical results, which are detailed in the following subsections, are encouraging as they show that our approach achieves higher expressivity and outperforms other related systems.

5.1. Experiment 1: Positive rules

In this experiment, we select C_SPARQL as a comparable system to handle positive rules. We do not consider CQELS [23] because its processing mode does not allow certain positive rules to be expressed: both *PR* and C_SPARQL process streaming data in batches while CQELS processes every new data item immediately and therefore cannot reason about elements appearing in the same window. We compare *PR* against C_SPARQL by using the well-known stream processing benchmark CityBench [1]. In particular we use query Q1, Q2, and Q10 as representative samples in terms of number of query patterns and presence of join operators. Details of those queries are available in the CityBench github[5]. To make sure that both engines return the same result format (triple) for a fair comparison, we modify the SELECT statement in both queries to a CONSTRUCT statement, and we refer to them as to Q1C, Q2C, and Q10C respectively. We translate queries Q1C, Q2C, and Q10C into ASP positive rule sets for *PR*. We refer to those rule sets as R1C, R2C, and R10C respectively. Listing 2 shows the rule set obtained by translating Q1C. We evaluate latency and memory consumption of the two engines by increasing the input streaming rate. The streaming rate can be changed by changing the frequency parameter in the CityBench configuration. We stream data for 10 minutes with two different frequencies $f = 1$ and $f = 2$. Results shown in Figure 7 and Figure 9 indicate that the latency for *PR* is minimal compared to C_SPARQL in both frequencies and queries. More specifically, *PR* performs almost 3 times (or more) faster than C_SPARQL for queries Q1C, Q2C in the case of frequency $f = 1$ (or $f = 2$, respectively). For query Q10C, the performance of both *PR* and C_SPARQL remains the same in both $f = 1$ and $f = 2$. Also, it is noticeable that the memory consumption of *PR* is less than a half of C_SPARQL memory consumption (see Figure 8 and Figure 10). Notice that with those queries in CityBench, the input dependency graph is strongly connected (there is an edge between any two vertexes), therefore the parallel optimization cannot be exploited. It represents that our reasoner outperforms C-SPARQL engine without triggering its parallel optimization mode.

5.2. Experiment 2: Recursive positive rules

For the experiment with recursive positive rules that are not supported by C_SPARQL, we compare *PR* against *R* and Jena reasoner[6] by using a widely used benchmark for reasoning systems, the Lehigh University Benchmark (LUBM [21]). We selected a different benchmark for experiment 2 due to limitations regarding expressivity of rules in CityBench. In order to evaluate these engines, we create a set of rules as in Listing 3 which includes 4 recursive rules over 15 rules. We use Univ-Bench Artificial Data Generator[7] to generate and stream data to the engines. Due to the fact that the Jena reasoner does not support data stream process-

[5] https://github.com/CityBench/Benchmark
[6] https://jena.apache.org/documentation/inference/
[7] https://github.com/rvesse/lubm-uba

```
observerBy(ObId) :- ssn_observedBy(ObId, "_AarhusTrafficData182955").
observerBy(ObId) :- ssn_observedBy(ObId, "_AarhusTrafficData158505").
_result(ObId, V) :- observerBy(ObId), sao_hasValue(ObId, V),
                    ssn_observedProperty(ObId, P), rdf_type(P, "ct_CongestionLevel").
```

Listing 2: Rules translated from query Q1 in CityBench

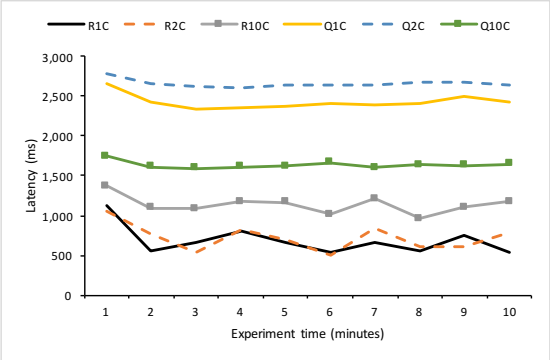

Fig. 7. Latency (f = 1)

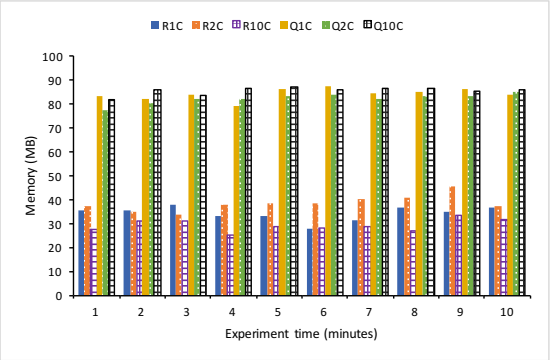

Fig. 8. Memory consumption (f =1)

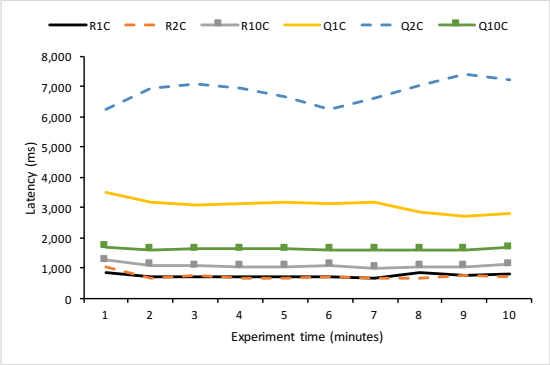

Fig. 9. Latency (f = 2)

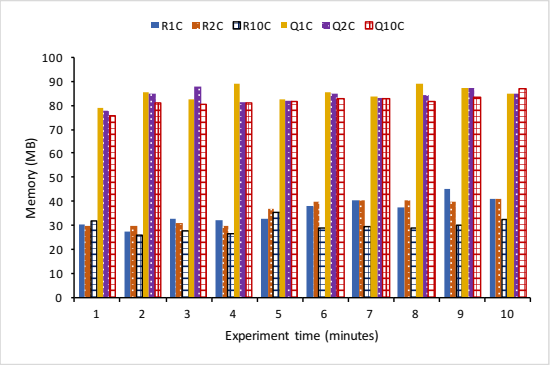

Fig. 10. Memory consumption (f =2)

ing, we run this experiment in two settings: static and streaming.

Static setting. In this setting, we evaluate *PR*, *R* and Jena reasoner with different sizes of input data from 5k to 100k (k=1000) triples. We trigger each engine 3 times per each input data size and take the average. Figure 11 and Figure 12 show the effect over latency and memory consumption with increasing number of triples for the three engines. A closer look at the results in Figure 11 reveals that *PR* outperforms *R* over subsequent increase from 10k to 100k (*R* can not process 60k and 100k triples). Compare to Jena, *PR* is slightly slower when the input size is smaller than 30k. However, *PR* is considerably faster than Jena when the number of triples is bigger than 30k. When the input size increases from 60k to 100k triples, the latency of Jena increases sharply from 200 seconds to 750 seconds while *PR*'s latency only increases slightly from 100 seconds to 200 seconds. This is an indication of the scalability of our approach over increasing size of the input. For memory consumption, Figure 12 shows that all engines have increasing memory consumption issue but Jena seems to be better at memory management when increasing the number of input triples.

Streaming setting. In the streaming setting, we trigger *PR* and *R* by streaming triples for 10 minutes with various rates from 1k to 5k triples/second. We use the time-based window size of 3 seconds with the sliding step of 2 seconds. Figure 13 reports latency observed from *PR* and *R*. It shows that *PR* performs as *R* at

```
#prefix rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
#prefix uniben : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>.

(r1) rdf_type(X,"Profesor") :- rdf_type(X, "uniben_FullProfessor").
(r2) rdf_type(X,"Profesor") :- rdf_type(X, "uniben_AssociateProfessor").
(r3) rdf_type(X,"Profesor") :- rdf_type(X, "uniben_AssistantProfessor").
(r4) canBecomeDean(X,U) :- rdf_type(X,"Professor"), uniben_worksFor(X,D),
                           uniben_subOrganizationOf(D,U).
(r5) canBecomeHeadOf(X,D) :- uniben_worksFor(X,D).
(r6) commonResearchInterests(X,Y):- uniben_researchInterest(X,R),
                           uniben_researchInterest(Y,R).
(r7) commonPulication(X,Y):- uniben_publicationAuthor(P,X),
                           uniben_publicationAuthor(P,Y).
(r8) commonResearchInterests(X,Y) :- commonPulication(X,Y).
(r9) uniben_teacherOf(Y,C):- commonResearchInterests(X,Y), uniben_teacherOf(X,C).
(r10) commonResearchInterests(X,Y):- uniben_advisor(X,Z), uniben_advisor(Y,Z).
(r11) canRequestRecommendationLetter(X,Z) :- uniben_advisor(X,Z).
(r12) canRequestRecommendationLetter(X,Z) :- teaches(Z,X).
(r13) teaches(X,Y):- uniben_teacherOf(X,C), uniben_takesCourse(Y,C).
(r14) teaches(X,Y):- uniben_teachingAssistantOf(X,C), uniben_takesCourse(Y,C).
(r15) suggestAdvisor(X,Y):- teaches(Y,X).
```

Listing 3: A set of ASP rules inspired from LUBM

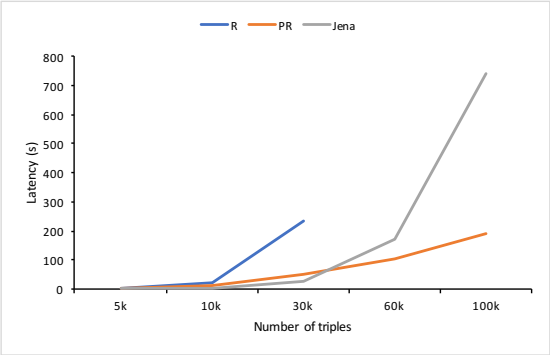

Fig. 11. Latency (recursive rules with static setting)

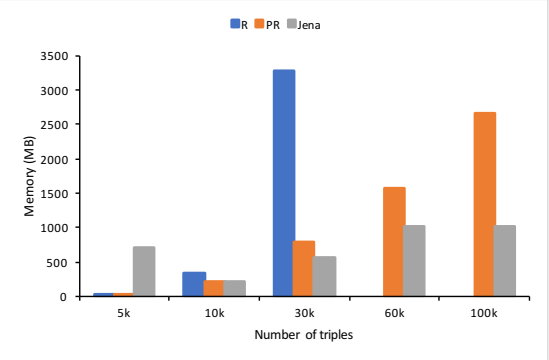

Fig. 12. Memory consumption (recursive rules with static setting)

the streaming rate of 1k triples/second. The reason for this is that the number of input triples is small enough and the Clingo solver does not suffer from exponential grounding. However, we observe a benefit of parallel optimization in *PR* at the streaming rates of 3k and 5k triples/second where *PR* performs much faster than *R*. In addition, the latency of *PR* is more stable than the one of *R* during the 10-minute streaming. This means that our approach generates a smaller ground program and a smaller search space, speeding up both grounding and solving of the reasoner. For memory consumption that is illustrated in Figure 14, *PR* consumes slightly less memory than *R*. The figures also shows that there is a considerable increase in memory consumption when streaming rate increases from 1k to 5k triples/seconds.

5.3. *Experiment 3: Stratified negation rules*

We now focus on a rule set which has stratified negations. We modify rules r_5, r_{12} and r_{15} in the rule set of experiment 2 with 3 negation-as-failure atoms as in Listing 4. As a result, the experimental rule set now includes 4 recursive rules and 3 negation-as-failure rules over 15 rules. We compare *PR* against *R* only since the Jena reasoner does not support negation-as-failure.

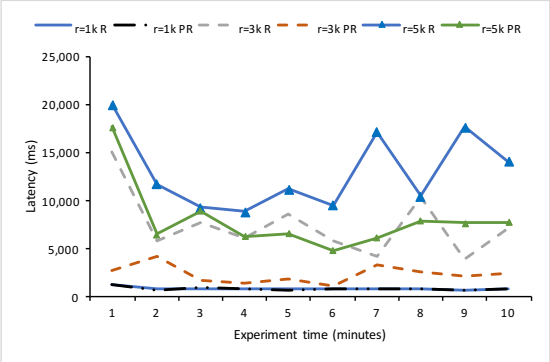

Fig. 13. Latency (recursive rules with streaming setting)

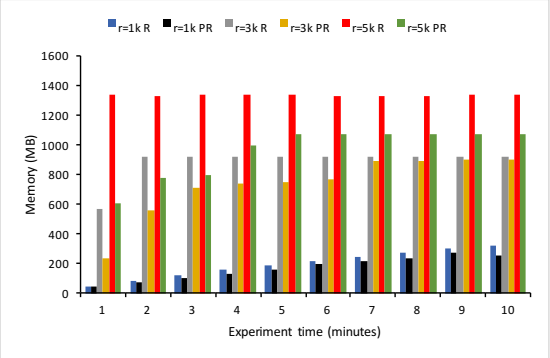

Fig. 14. Memory consumption (recursive rules with streaming setting)

Similar to experiment 2, we evaluate the same two engines for 10 minutes with various streaming rates from 1k to 5k triples/second. Figure 15 and Figure 16 illustrate a similar pattern in latency and memory consumption as observed in the experiment 2. *PR* has faster reasoning time at streaming rates 3k and 5k triples/second, but consumes slightly higher memory compared to *R* at 5k triples/seconds.

6. Related Works

Parallel strategies were important features of database technology in the nineties in order to speed up the execution of complex queries [9]. In Semantic Web, the parallelism in reasoning has been studied in [15, 25, 28–30] where a set of machines is assigned a partition of the parallel computation. [15] presents a distributed ontology reasoning and querying system which employs Distributed Hash Table method to organize the instance data. [25] 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, [30] proposes a technique for materializing the closure of an RDF graph based on MapReduce [11]. The authors in [29] 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 based on the well-founded semantics [28]. While the works in [15, 25, 30] focus on monotonic reasoning, [28, 29] examine non-monotonic reasoning over massive data. However, these attempts do not consider the streaming setting and do not rely on the stable model semantics.

In ASP, several works about parallel techniques for the evaluation of a logic program have been proposed [4, 10, 17, 20, 26], focusing on both phases of the ASP computation, namely grounding and solving. Concerning the parallelization of the grounding phase, the work in [4] 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. [10] explores some structural properties of the input program via the defined dependency graph in order to detect subprograms that can be evaluated in parallel. [26] 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, [20] 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. [17] introduces a conflict-driven algorithm to compute the answer sets based on constraint processing and satisfiability checking. In short, [4, 10, 26] focus on parallel instantiation by splitting a logic program in order to obtain a smaller ground program, [17, 20] compute the answer sets from that ground program in parallel. These approaches have been implemented in state-of-the-art ASP solvers such as Clingo and DLV. In this paper, we are not partitioning the logic program. We are fo-

```
(r'5)  canBecomeHeadOf(X,D) :- uniben_worksFor(X,D), uniben_headOf(Z,D),
                               not commonResearchInterests(X,Z).
(r'12) cannotRequestRecommendationLetter(X,Z) :- teaches(Z,X),not uniben_advisor(X,Z).
(r'15) suggestAdvisor(X,Y):- teaches(Y,X),not uniben_advisor(X,Z).
```

Listing 4: Negation-as-failure rules

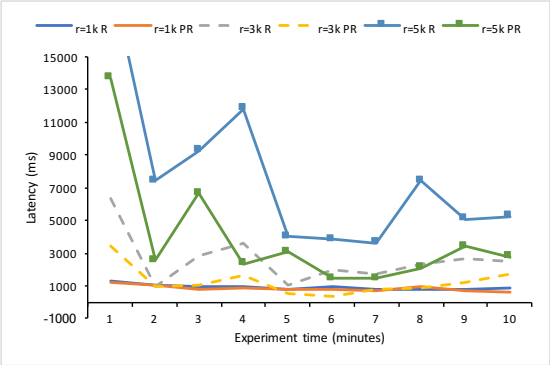

Fig. 15. Latency (recursive and stratified negation rules)

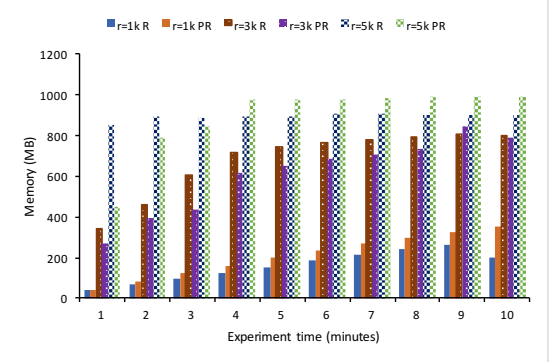

Fig. 16. Memory consumption (recursive and stratified negation rules)

cusing instead on partitioning the input and evaluating each partition on a different copy of the whole program with the intuition that this approach is data-driven and can result in a faster run-time analysis since it does not consider the whole program in any case, but only the rules that are triggered based on the (partitioned) streaming input.

A different approach to enhance the scalability of expressive stream reasoning is based on incremental methods. There are two reasoners proposed recently based on the LARS framework [7], namely Ticker [8] and Laser [6]. Ticker translates the plain LARS (more specifically, a fragment of LARS) to ASP and supports two reasoning strategies: one utilizes Clingo with a static ASP encoding and the other applies truth maintenance techniques to adjust models incrementally. Similarly, Laser also relies on incremental model update to avoid unnecessary re-computations by annotating formula with two time markers. However, this engine restricts its logic programs to a stratified tractable fragment of LARS to ensure the uniqueness of models.

7. Conclusion and Future Work

Scalability is a key challenge for the applicability of reasoning techniques to rapidly changing information. In this paper, we consider the challenge of creating new semantic knowledge from diverse and dynamic data for complex problem solving, and doing that in a scalable way. To address this challenge, we focus on an approach that leverages semantic technologies to integrate and pre-process RDF streams on one side, and expressive inference enhanced with parallel execution on the other side.

Building upon previous work, and following up on our initial investigation of the trade-off between scalability and expressivity of rule-based reasoning over streaming RDF data, in this paper we provided a clear characterization and formal definition of our approach to parallelization of stream reasoning by input dependency analysis (both at the predicate and at the atom level) that was first introduced in [27]. We implemented the proposed approach as an extension of the StreamRule reasoner and provided a proof of correctness under the assumption that no recursion through negation is present in the rules, thus guaranteeing the uniqueness of the solution. Furthermore, we considered the different levels of expressivity that are supported by the reasoning layer of our prototype implementation and conducted a detailed experimental evaluation by comparison with different systems based on their expressivity. This evaluation indicates that our reasoner not only has a competitive performance in comparison with existing systems but it also supports

higher expressivity of reasoning tasks. This work is also a demonstration that expressive reasoning is possible also in streaming environments, and it paves the way for investigating feasible solutions in this space.

Our performance evaluation demonstrates that the combination of RDF Stream Processing and ASP-based reasoning for heterogeneous and highly dynamic data is possible and promising, even when recursion and default negation are used, and that the performance does not degrade for simpler tasks, thus being comparable with alternative systems.

Stream reasoning is a new and active area of research within Semantic Web, Knowledge Representation and Reasoning community and there are many open questions and interesting directions for investigation that we are currently working on as next steps, we discuss a few in the remainder of this section.

In order to avail the full power of ASP-based reasoning, the ability to generate multiple solutions is key, but this requires a deeper investigation of how correctness can be maintained when partitioning and merging results in the presence of multiple answer sets. This is a key step we are currently exploring to exploit the full expressivity of ASP-based reasoning for semantic streams.

Another direction for investigation is related to the definition of multiple heuristics for splitting the graph and duplicating nodes. Our current solution is based on finding and merging cliques using a threshold score on the ratio between common and different vertexes, to decide where to split and duplicate. Different heuristics that also consider the size of the cliques and that aim at load balancing would contribute to the overall performance of the system. Leveraging information about the distribution of ground atoms across the different predicates could also be a good information to design better heuristics and for load balancing. This could also inform the current partitioning function so that the splitting process does not rely on predicate-level analysis only. We believe this can have an important effect on computation time that needs to be further investigated.

Despite incremental evaluation and parallel execution are different ways of tackling the scalability issue, we believe a comparison with these systems in terms of expressivity vs. scalability trade-off will enable us to share important insights for future work and advances in the Stream Reasoning field. Therefore, another part of our ongoing work is to perform an extensive evaluation aimed at comparing our reasoner with Ticker and Laser. To do so, we are currently building a benchmark for ASP-based stream reasoning which builds upon state-of-the-art static ASP benchmarking [18] and RDF stream processing benchmarks (e.g., [1], [22]). Our resulting benchmark is designed to cover various expressivity levels of complex reasoning and will support configurable parameters (e.g., input streaming rate, window size) for evaluating the behavior of the stream reasoners.

References

[1] M. I. Ali, F. Gao, and A. Mileo. CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets. In M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, editors, *International Semantic Web Conference*, pages 374–389. Springer, Cham, 2015. DOI: 10.1007/978-3-319-25010-6_25.

[2] 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*, WWW '11, pages 635–644. ACM, New York, USA, 2011. DOI: 10.1145/1963405.1963495.

[3] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012. DOI: 10.3233/SW-2011-0053.

[4] 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. DOI: 10.1016/j.parco.2005.03.004.

[5] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 1061–1062. ACM, New York, USA, 2009. DOI: 10.1145/1526709.1526856.

[6] H. R. Bazoobandi, H. Beck, and J. Urbani. Expressive stream reasoning with Laser. In C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, and J. Heflin, editors, *International Semantic Web Conference*, pages 87–103. Springer, Cham, 2017. DOI: 10.1007/978-3-319-68288-4_6.

[7] H. Beck, M. Dao-Tran, and T. Eiter. LARS: A logic-based framework for analyzing reasoning over streams. In A. M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen, and J. Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, pages 87–93. Springer, Cham, 2018. DOI: 10.1007/978-3-319-73117-9_6.

[8] H. Beck, T. Eiter, and C. Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming*, 17(5-6):744–763, 2017. DOI: 10.1017/s1471068417000370.

[9] 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. DOI: 10.1007/bf01264013.

[10] F. Calimeri, S. Perri, and F. Ricca. Experimenting with parallelism for the instantiation of ASP programs. *Journal of Algorithms*, 63(1):34–54, 2008. DOI: 10.1016/j.jalgor.2008.02.003.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. DOI: 10.1145/1327452.1327492.

[12] T. M. Do, S. W. Loke, and F. Liu. Answer set programming for stream reasoning. In C. Butz and P. Lingras, editors, *Advances in Artificial Intelligence*, pages 104–109. Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-21043-3_13.

[13] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, and R. A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, pages 40–110. Springer, Berlin, Heidelberg, 2009. DOI: 10.1007/978-3-642-03754-2_2.

[14] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. DLVhex: A prover for Semantic Web reasoning under the answer set semantics. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 1073–1074. IEEE, 2006. DOI: 10.1109/WI.2006.64.

[15] Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable distributed ontology reasoning using DHT-based partitioning. In J. Domingue and C. Anutariya, editors, *Asian Semantic Web Conference*, pages 91–105, Berlin, Heidelberg, 2008. Springer. DOI: 10.1007/978-3-540-89704-0_7.

[16] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Answer set programming for stream reasoning. *CoRR, abs/1301.1392*, 2013. http://adsabs.harvard.edu/abs/2013arXiv1301.1392G.

[17] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012. DOI: 10.1016/j.artint.2012.04.001.

[18] M. Gebser, M. Maratea, and F. Ricca. The design of the seventh answer set programming competition. In M. Balduccini and T. Janhunen, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 3–9. Springer, Cham, 2017. DOI: 10.1007/978-3-319-61660-5_1.

[19] S. Germano, T.-L. Pham, and A. Mileo. Web stream reasoning in practice: on the expressivity vs. scalability tradeoff. In B. ten Cate and A. Mileo, editors, *Web Reasoning and Rule Systems*, pages 105–112. Springer, Cham, 2015. DOI: 10.1007/978-3-319-22002-4_9.

[20] J. Gressmann, T. Janhunen, R. E. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A platform for distributed answer set solving. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 227–239. Springer, Berlin, Heidelberg, 2005. DOI: 10.1007/11546207_18.

[21] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005. DOI: 10.1016/j.websem.2005.06.005.

[22] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *International Semantic Web Conference*, pages 300–312. Springer, Berlin, Heidelberg, 2012. DOI: 10.1007/978-3-642-35173-0_20.

[23] D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy, and E. Blomqvist, editors, *The Semantic Web – ISWC 2011*, pages 370–388. Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-25073-6_24.

[24] A. Mileo, A. Abdelrahman, S. Policarpio, and M. Hauswirth. StreamRule: a nonmonotonic stream reasoning system for the Semantic Web. In W. Faber and D. Lembo, editors, *Web Reasoning and Rule Systems*, pages 247–252. Springer, Berlin, Heidelberg, 2013. DOI: 10.1007/978-3-642-39666-3_23.

[25] 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. DOI: 10.1016/j.websem.2009.09.002.

[26] 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. DOI: 10.1017/S1471068411000652.

[27] T.-L. Pham, A. Mileo, and M. I. Ali. Towards scalable nonmonotonic stream reasoning via input dependency analysis. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1553–1558. IEEE, 2017. DOI: 10.1109/ICDE.2017.226.

[28] I. Tachmazidis, G. Antoniou, and W. Faber. Efficient computation of the well-founded semantics over big data. *Theory and Practice of Logic Programming*, 14(4-5), 2014.

[29] I. Tachmazidis, G. Antoniou, G. Flouris, and S. Kotoulas. Towards parallel nonmonotonic reasoning with billions of facts. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'12, pages 638–642. AAAI Press, 2012. http://dl.acm.org/citation.cfm?id=3031843.3031926.

[30] J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen. Scalable distributed reasoning using MapReduce. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, editors, *Proceedings of the 8th International Semantic Web Conference*, pages 634–649. Springer, Berlin, Heidelberg, 2009. DOI: 10.1007/978-3-642-04930-9_40.