# Handling Impossible Derivations during Stream Reasoning

Hamid R. Bazoobandi, Henri Bal, Frank van Harmelen, and Jacopo Urbani

Vrije Universiteit Amsterdam, The Netherlands
h.bazoubandi@vu.nl,[bal,frankh,jacopo]@cs.vu.nl

**Abstract.** With the rapid expansion of the Web and the advent of the Internet of Things, there is a growing need to design tools for intelligent analytics and decision making on streams of data. Logic-based frameworks like LARS allow the execution of complex reasoning on such streams, but it is paramount that the computation is completed in a timely manner before the stream expires. To reduce the runtime, we can extend the validity of inferred conclusions to the future to avoid repeated derivations, but this is not enough to avoid all sources of redundant computation. To further alleviate this problem, this paper introduces a new technique that infers the impossibility of certain derivations in the future and blocks the reasoner from performing computation that is doomed to fail anyway. An experimental analysis on microbenchmarks shows that our technique leads to a significant reduction of the reasoning runtime.

## 1 Introduction

In highly dynamic environments like the Web or the Internet of Things, there are many use cases that require an efficient processing of large streams of data to provide complex data analytics or intelligent decision making. For instance, the content of the stream can be used to make predictions about future behaviors (e.g., financial market movement), or to build an accurate representation of the current environment (e.g., crowd control).

In some cases, a semantic-oriented approach is needed to process the stream. An example is given by autonomous driving, which is currently one of the most prominent frontiers of AI. As it was recently shown by Suchan et al. [31], there are situations that cannot (yet) be handled by deep-learning-based computer vision techniques, and this can lead to safety concerns. The *occlusion scenario* is an example of such a situation. This scenario occurs when another vehicle, which is clearly visible in close proximity, suddenly disappears and reappears shortly after (e.g., due to the steering of a third vehicle). When this event occurs, a system that relies only on the input provided by computer vision might erroneously conclude that the vehicle is no longer in close proximity, and consequently act on this false premise. Humans, in contrast, (usually) apply some logic-based reasoning and conclude that the vehicle is still nearby although it is hidden.

Suchan et al. mention this scenario to motivate the need for semantics and logic-based reasoning of temporal data. Currently, one of the most prominent frameworks for this type of processing is LARS [5]. LARS is ideal for use cases like the aforementioned one. First, its semantics is grounded on Answer Set Programming (ASP); thus it provides an AI that is explainable by design, which means that it can be used also by non experts or audited by regulators. Second, LARS offers a variety of operators that are specifically designed for modeling streams to allow the execution of complex reasoning without making it harder than it is in ASP. For instance, LARS offers window operators that allow the restriction of the analysis to the last data in the stream, or other operators like @ which specifies when a derived conclusion will be valid.

Since often the data in the stream expires after a short amount of time, it is paramount that reasoning is performed in a timely manner. Recently, several works have used LARS to implement stream reasoning that reconciles the expressivity of LARS with high performance. One of such reasoner is *Ticker* [6], while a more recent distributed implementation is presented in [14]. Another of such reasoners is *Laser*, which we presented in a previous paper [4]. Laser distinguishes itself from the previous two by focusing on a smaller and more tractable fragment of LARS called *Plain LARS*. Another distinctive feature is that Laser introduces a new technique that annotates the formulae with two timestamps, called *consideration* and *horizon* times, to extend the validity of the formulae in the future to avoid that they are re-derived at each time point. This technique is particularly effective when the body of the rules contains the LARS operators $\diamond$ (validity at *some* time point) or @ (validity at *one specific* time point), and it can lead to significantly faster runtimes. However, this technique does not work with the operator $\Box$ (validity at *all* time points) because the semantics of this operator is such that the validity cannot be guaranteed in the future.

In this paper, we present a new technique to further limit the number of redundant derivations. Our technique targets formulae for which the consideration and horizon timestamps are not effective (i.e., the rules that use the $\Box$ operator). The main idea is to identify the cases when it will be impossible to produce some derivations in the future and to use this knowledge to disable rules that won't be able to produce any new conclusion. For example, consider the LARS rule $\boxplus^3\Box p(a) \to q(a)$. This rule specifies that if the fact $p(a)$ appears in the stream in last three time points, then we can infer $q(a)$. In this case, if the stream does not contain $p(a)$ at the current time point, then we can conclude that for the next three time points the rule will never be able to infer $q(a)$, thus making it an "impossible derivation". Since we know that it is impossible that $q(a)$ will be derived, we can disable the rule and simplify reasoning. Moreover, if other rules use $q(a)$ in their body, then they can also be disabled, with a further improvement of the performance.

We have implemented our technique in a new reasoner called *Laser2*, which is a completely rewritten Plain LARS reasoner in Golang. Our experiments show that our technique returns significant improvements in terms of runtime. The code of Laser2 and other evaluation data can be found at at `https://bitbucket.org/hrbazoo/laser`.

## 2 Background

We start our discussion with some background notions on logic programming and LARS [5]. Let $\mathcal{C}, \mathcal{V}, \mathcal{P}$ be disjoint sets of *constants*, *variables*, and *predicates*. A predicate $p$ can be either *extensional* or *intensional* and it is associated to a fixed arity $\mathsf{ar}(p) \geq 0$. A *term* is either a constant or variable. An *atom* is an expression of the form $p(\mathbf{t})$ where $p$ is a predicate, $\mathbf{t} = t_1, \ldots, t_n$ is a list of terms and $n = \mathsf{ar}(p)$. A *ground* expression is an expression without any variable. A *fact* is a ground atom.

Let $\mathcal{A}$ be the set of all facts we can construct from $\mathcal{P}$ and $\mathcal{C}$ and let $\mathcal{A}^{\mathcal{E}} \subseteq \mathcal{A}$ be the subset of facts with extensional predicates. A *timeline* $\mathbf{T}$ is a closed non-empty interval in the set of natural numbers $\mathbb{N}$. We refer to each member in a timeline as a *time point*. Abusing notation, we write $t \in \mathbf{T}$ to indicate a generic time point in $\mathbf{T}$. We are now ready to define the notion of *stream*.

**Definition 1.** *A* stream $S = (\mathbf{T}, v)$ *is a pair of a timeline and* evaluation function $v : \mathbb{N} \mapsto 2^{\mathcal{A}}$*, which maps integers to set of atoms in* $\mathcal{A}$ *with the constraint that* $v(t) \mapsto \emptyset$ *for each* $t \notin \mathbf{T}$*.*

Intuitively, $v$ is used to map time points to sets of facts. We say that $S$ is a *data stream* if $v$ maps only to atoms in $\mathcal{A}^{\mathcal{E}}$. Also, a stream is *ground* if $v$ maps only to facts. Finally, we say that $S' = (\mathbf{T}, v')$ is a *substream* of $S = (\mathbf{T}, v)$, denoted as $S' \subseteq S$, if $v'(t) \subseteq v(t)$ for each time point in $\mathbf{T}$. A *window function* $w$ is a computable function which receives in input a stream $S$ and a time point $t$ and returns in output a stream $S' \subseteq S$. LARS proposes several window functions: a *time-based* window function $w_n$ returns a substream that filters out all the atoms that are not in $t$ or in the previous $n-1$ time points; a *tuple-based* window function returns a substream with the last $n$ facts, etc. In this paper, we consider only time-based windows functions, and leave an extension of our technique to other types of window functions as future work.

In this paper, we focus on a fragment of LARS called *Plain LARS* [4]. Plain LARS restricts some features of LARS in order to enable a fast computation. In Plain LARS, an *extended atom* $\alpha$ is a formula that complies with the grammar

$$\alpha ::= a \mid @_t a \mid \boxplus^n @_t a \mid \boxplus^n \Diamond a \mid \boxplus^n \Box a$$

where $t \in \mathbb{N}$, $a$ is an atom, @ is an operator that specifies that $a$ holds at $t$, $\boxplus^n$ is used to restrict the stream using the time-based window $w_n$, $\Diamond$ states that $a$ should hold at least in one time point, while $\Box$ states that $a$ should hold at every time point. A (ground) *rule* is an expression of the form:

$$B_1 \land \ldots \land B_m \to H \tag{1}$$

where $B_1, \ldots, B_m$ are (ground) extended atoms, and $H$ is a (ground) extended atom that is either an atom or of the form $@_t a$. A (ground) *program* is a finite set of (ground) rules. Let $r$ be a rule as shown in (1). Throughout, we use the shortcut $\mathsf{B}(r)$ (body) to refer to the left-side of the rule and $\mathsf{H}(r)$ (head) for the right-side.

We first define the semantics for ground programs. Let $M = \langle S, w_n, \mathcal{B} \rangle$ be a structure where $S = (\mathbf{T}, v)$ is a ground stream of facts in $\mathcal{A}$, $w_n$ is the time-based window function, $\mathcal{B} \subseteq \mathcal{A}$ is a set of facts called *background knowledge*. Then, $M$ *entails* $\alpha$ at time point $t$, denoted as $M, t \Vdash \alpha$, as follows:

$$\begin{array}{llllll}
\text{if} & \alpha = a & \text{then} & M, t \Vdash \alpha & \text{iff} & a \in v(t) \text{ or } a \in \mathcal{B}, \\
\text{if} & \alpha = \Diamond a & \text{then} & M, t \Vdash \alpha & \text{iff} & M, t' \Vdash a \text{ for some } t' \in \mathbf{T}, \\
\text{if} & \alpha = \Box a & \text{then} & M, t \Vdash \alpha & \text{iff} & M, t' \Vdash a \text{ for all } t' \in \mathbf{T}, \\
\text{if} & \alpha = @_{t'} a & \text{then} & M, t \Vdash \alpha & \text{iff} & M, t' \Vdash a \text{ and } t' \in \mathbf{T}, \\
\text{if} & \alpha = \boxplus^n \beta & \text{then} & M, t \Vdash \alpha & \text{iff} & M', t \Vdash \beta \text{ where } M' = \langle w_n(S, t), w_n, \mathcal{B} \rangle, \\
\text{if} & \alpha = \wedge_{i=1}^m B_i & \text{then} & M, t \Vdash \alpha & \text{iff} & M, t \Vdash B_i \text{ for all } 1 \leq i \leq m, \\
\text{if} & \alpha = B \rightarrow H & \text{then} & M, t \Vdash \alpha & \text{iff} & M, t \nVdash B \vee M, t \Vdash H.
\end{array}$$

Given a data stream $D = (\mathbf{T}, v_D)$, we say that $M$ is a *model* of $P$ (for $D$) at time point $t$, denoted as $M, t \models P$, if $M, t \Vdash r$ for every $r \in P$ and $M$ and $S$ coincides with $D$ on $\mathcal{A}^{\mathcal{E}}$, i.e., $S \supseteq D$ and every fact with extensional predicate in $S$ at time point $x$ is also in $D$ at $x$. If no other model $M' = \langle S', w_n, \mathcal{B} \rangle \neq M$ exists such that $S' = (\mathbf{T}, v')$ and $v'(t) \subseteq v(t)$ for any $t \in \mathbf{T}$, then $M$ is *minimal*.

The semantics of a non ground program $P$ (i.e., a program where some rules contain variables) equals to the semantics of the ground program that is obtained by grounding all rules in $P$ with all possible substitutions in $\mathcal{C}$. For instance, if $\mathcal{C} = \{c_1, c_2\}$ and $P = \{p(X) \rightarrow q(X)\}$ where $X$ is a variable, then $M, t \models t$ iff $M, t \models P'$ where $P' = \{p(c_1) \rightarrow q(c_1), p(c_2) \rightarrow q(c_2)\}$. Given an input data stream and a program, our goal is to compute *answer streams* and return the derivations to the user.

**Definition 2.** *Stream $S$ is an* answer stream *of program $P$ for data stream $D$ at time point $t$ if $M = \langle S, w_n, \mathcal{B} \rangle$ is a minimal model of the reduct $P^{M,t} = \{r \in P \mid M, t \models \mathsf{B}(r)\}$.*

We have now all the elements to define the output of our computation.

**Definition 3.** *Let $S = (\mathbf{T}, v)$ be the answer stream of program $P$ (for $D$) at time point $t$. Then, the* output *is the set $v(t) \setminus \mathcal{A}^{\mathcal{E}}$, that is, the set of all the atoms with intensional predicates that can be inferred by $P$ at $t$.*

## 3 Intuition

The example below illustrates the computation performed during LARS reasoning and is useful to provide an intuitive description of our technique.

**Example 1.** Let $\mathcal{P} = \{highTemp, warning, error, shutdown\}$ be a set of predicates where only $highTemp$ is extensional and $\mathcal{C} = \{b_1, b_2\}$. We consider an input stream $D = (\mathbf{T}, v)$ which is defined with the timeline $\mathbf{T} = \langle 1, \ldots, 15 \rangle$ and

$$v = \{2 \mapsto \{highTemp(b_1), highTemp(b_2)\}, 3 \mapsto \{highTemp(b_2)\}\},$$

that is, a high temperature is observed only at time points 2 and 3 (all other time points are mapped to the empty set).

Moreover, let us consider a ground program $P$ with the rules

$$\boxplus^{10}\Diamond highTemp(b_1) \rightarrow warning(b_1) \qquad (r_1)$$

$$\boxplus^{3}\Box highTemp(b_2) \rightarrow error(b_2) \qquad (r_2)$$

$$error(b_2) \rightarrow shutdown(b_2) \qquad (r_3)$$

Given this input, an answer stream of $P$ for $D$ at time point 2 is the stream $S_2 = (\mathbf{T}, v')$ where

$$v' = \{2 \mapsto \{highTemp(b_1), highTemp(b_2), warning(b_1)\}, 3 \mapsto \{highTemp(b_2)\}\}$$

while $S_3 = (\mathbf{T}, v'')$ where

$$v'' = \{2 \mapsto \{highTemp(b_1), highTemp(b_2)\}, 3 \mapsto \{highTemp(b_2), warning(b_1)\}\}$$

is an answer stream at time point 3. In this case, the output will be the set $\{warning(b_1)\}$ both at time point 2 and 3.

Since the output of a LARS program is defined with respect to a single time point, the framework does not put any restriction on the order in which the time points should be considered. In Example 1, for instance, a user could decide to compute first the output at time point 3 and then at time point 2. In practice, however, streams are typically evaluated time point after time point.

This evaluation criterion can be exploited to avoid triggering redundant derivations. In Example 1, a naïve application of rule $r_1$ will derive $warning(b_1)$ twice; both at time point 2 and 3. However, the second derivation can be avoided since we know that $warning(b_1)$ will hold at least until time point 12 because $r_1$ fires if $highTemp(b_1)$ appears at least once in the last 10 time points.

In [4], it has been shown how we can exploit this observation by annotating the formulae with two timestamps: a *consideration* and a *horizon* time. The consideration time identifies the first time point where the formula holds, while the horizon time identifies the last time point where the formula is guaranteed to hold. For instance, at time point 2 the formula $\boxplus^{10}\Diamond highTemp(b_1)$ is annotated with a consideration time equals to 2 (i.e., the first time point where this formula is inferred). Instead, the horizon time equals to 12 since we know that the body of the rule will hold until 12. The annotated formula with these timestamps is denoted as $\boxplus^{10}\Diamond highTemp(b_1)_{[2,12]}$. From these annotations, it also follows that the fact $warning(b_1)$ can be annotated as $warning(b_1)_{[2,12]}$. Since these two formulae will hold in the future, they are kept in the working memory until the current time point is greater than the horizon time. When this occurs, they expire and can be removed.

When we execute a rule, we can use the annotations to perform a check that is similar to the one of Semi Naïve Evaluation (SNE) [1] – a well-known Datalog technique to reduce the number of duplicate derivations. The idea behind SNE is to block the firing of the rule if no atom that instantiates the body was derived in the previous step. In our setting, we can apply a similar principle and enforce that at least one formula used in the body has a consideration time equal to the current time point. In our example, this constraint will block the application of $r_1$ at time point 3 because $\boxplus^{10}\Diamond highTemp(b_1)_{[2,12]}$ has already been considered.

While the consideration and horizon timestamps are useful to reduce the runtime [4], their introduced benefit cannot be extended to formulae that use the operator $\Box$. In fact, a formula like $\Box a$ holds only if $a$ holds at *every* time point. Because of this constraint, we are unable to guarantee that it will hold in the future, hence we cannot extend the horizon time.

The technique presented in this paper aims precisely at overcoming this limitation. The main idea is the following: Although we cannot guarantee that a formula with $\Box$ will hold in the future, sometimes we can guarantee that the formula will *not* hold. Let us consider again Example 1. At time point 1, the absence of facts with the predicate *highTemp* in the data stream tells us that rule $r_2$ will never fire for at least the following three time points. Consequently, also $r_3$ will never fire and therefore can be safely ignored until time point 4. By doing so, our technique complements the usage of consideration and horizon times by covering the formulae where these two time stamps are not beneficial.

## 4 Formal Description

Algorithm 1 describes the reasoning procedure with our technique enabled to compute the output of a Plain LARS program. Function reason receives in input a data stream $D = (\mathbf{T}, v_D)$, background knowledge $\mathcal{B}$ and a program $P$ and returns the *output on* $\mathbf{T}$, i.e., a data structure ($Out$ in Algorithm 1) that contains the output at each time point in $\mathbf{T}$ ($Out[t_1]$ contains the output at time point $t_1$, $Out[t_2]$ contains the output at time point $t_2$, etc.). The presented algorithm assumes that the user is interested in computing the output at each time point. If this is not the case, then the algorithm can be easily adapted.

The computation of reason can be divided into four parts:

- *Init* (lines 1–7): In this phase the algorithm initializes various data structures;
- *EnableRules* (lines 9–11): Rules that were previously disabled are re-enabled;
- *Reasoning* (lines 12–14): Computes the derivations at a given time point;
- *DisableRules* (lines 15–23): Rules that won't fire in the future are disabled.

**Init.** The procedure uses four global variables. $P_A$ contains the active rules, i.e., that are considered during reasoning while $P_I$ contains the disabled rules. Initially, $P_A$ equals to $P$ while $P_I$ is empty (line 7). $R$ is a multimap used to collect the rules that can be invalidated for some time points in the future. We use $R$ to retrieve these rules after we observe that there are no facts derived in the current time point. These rules have a formula of the form $\boxplus^x \Box p(\mathbf{t})$ in their body. Let $r$ be such a rule. In this case, $R$ maps $p$ to one tuple of the form $\langle r, x\rangle$ which indicates that $r$ can be disabled for $x$ time points (line 5). The variable $S$ refers to another multimap that point to the rules that derive formulae with a given predicate. We use $S$ to decide whether the exclusion of a rule can trigger the exclusion of other ones.

**Algorithm 1:** $\mathsf{reason}(D, \mathcal{B}, P)$

```
Input       : data stream $D = (\mathbf{T}, v_D)$, background data $\mathcal{B}$, program $P$
Output      : Output on $\mathbf{T}$
Global vars : $P_A, P_I, R, S$
1  $R := \emptyset \quad S := \emptyset$
2  foreach $r \in P$
3      Let $q$ be the predicate used in $\mathsf{H}(r)$
4      $S[q] := S[q] \cup \{r\}$
5      foreach $\alpha \in \mathsf{B}(r)$ such that $\alpha := \boxplus^x \Box p(\mathbf{t})$ do $R[p] := R[p] \cup \langle r, x \rangle$
6  Let $\mathbf{T}$ be of the form $\langle t_1, \ldots, t_n \rangle$
7  $P_A := P \quad P_I := \emptyset \quad t_i := t_1$
8  while $t_i \leq t_n$ do
9      foreach $\langle r, t \rangle \in P_I$ and $t = t_i$
10         $P_A := P_A \cup \{r\}$
11         $P_I := P_I \setminus \{\langle r, t \rangle\}$
12     $Out[t_i] := \emptyset$
13     Compute answer stream $S = (\mathbf{T}, v)$ of $P_A$ for $D$ at $t_i$
14     $Out[t_i] := v(t_i) \setminus v_D(t_i)$
15     foreach $p \in \mathcal{P}$ that does not appear in $v(t_i)$
16         foreach $\langle r, t \rangle \in R[p]$ such that $r \in P_A$
17             $P_A := P_A \setminus \{r\}$
18             if $\langle r, y \rangle \in P_I$
19                 $P_I := P_I \setminus \{\langle r, y \rangle\}$
20                 $l := \mathsf{max}(t_i + t, y)$
21             else $l := t_i + t$
22             $P_I := P_I \cup \{\langle r, l \rangle\}$
23             $\mathsf{disable}(r, l, t_i)$
24     $t_i := t_i + 1$
25 end
26 return $Out$
```

**EnableRules.** The procedure considers each time point in a sequence (line 8). Before reasoning starts, it checks whether some rules that were previously disabled can be included again. To this end, the procedure considers all rules in $P_I$ which have expired, re-add them to $P_A$, and remove them from $P_I$ (lines 10–11).

**Reasoning.** Reasoning is computed in lines 12–14. First, it initializes the data structure $Out$. Then, it computes the answer stream according to Definition 2 and the corresponding output as specified in Definition 3. Note that these are computed using only the rules in $P_A$. Our method is agnostic to the procedure that is used to compute the derivations. In our implementation, we rely on the reasoning procedure specified in [4], that is the one that uses consideration and horizon timestamps, but one could in principle use any other routine, as long as it computes a valid answer stream.

**DisableRules.** After the answer stream is computed, we check whether some rules can be disabled. First, we identify all the predicates which do not appear in

**Algorithm 2:** $\mathsf{disable}(r_d, l, t_i)$

```
    Input    : $r_d$ is the rule that was deactivated, $l$ the length of the
               deactivation, $t_i$ is the current time point
    Output   : Modified $P_A$ and $P_I$
27  Let $q$ be the predicate used in $\mathsf{H}(r_d)$
28  if $|S[q]| = 1$
29      foreach $r \in P_A$
30          $rm :=$ false   $g := l$
31          foreach $\alpha \in \mathsf{B}(r)$
32              if $\alpha = q(\mathbf{t})$   $rm :=$ true
33              if $\alpha = \boxplus^n \Box q(\mathbf{t})$
34                  $g := \mathsf{max}(l, t_i + n)$
35                  $rm :=$ true
36          if $rm =$ true
37              $P_A := P_A \setminus \{r\}$
38              $P_I := P_I \cup \{\langle r, g \rangle\}$
39              $\mathsf{disable}(r, g, t_i)$
```

the output at the current time point (line 15). If there is a body atom with the operator $\Box$ in a rule in $P_A$ (line 16), then we remove the rule from $P_A$ (line 17) and add it to $P_I$ (line 22). When we add $r$ to $P_I$, we also specify the number of time points for which the rule should remain disabled. This number corresponds to the size of the window. If the rule is already disabled (this can occur if $r$ has multiple body atoms with $\Box$), then we use the maximum time point (line 20).

If a rule is disabled, then other rules can be disabled as well. To this purpose, we invoke the function $\mathsf{disable}$, reported in Algorithm 2. The function receives in input the rule that was just removed, i.e., $r_d$, the time point until $r_d$ will be disabled, and the current time point. First, we consider further rules only if $r_d$ is the only rule that derives facts with the predicate in the head ($q$, see line 28). If this occurs, then some rules that use $q$ in the body won't be able to fire as well. These are the rules where $q$ appears either as body atom or used with the $\Box$ operator (with other operators, the rule can still fire). We identify such rules in the loop in lines 31–35 with the flag $rm$. If the flag is enabled, then the rule is disabled until the time point $g$ (lines 36–39). Note that if the body atom is used inside a window, then $g$ is updated considering the maximum time point as expiration time point (line 34). After this, the procedure is invoked recursively (termination is ensured because in the worst case all rules in $P_A$ are removed and then the recursive call will not occur).

**Example 2.** Let us consider the input in Example 1. At time point 1, the stream is empty. Thus, $Out[1]$ will be equal to $v_D(1)$. Therefore, predicate $highTemp$ will be considered in the loop in line 16. The tuple $\langle r_2, 3\rangle$ is selected and $r_2$ is disabled by removing it from $P_A$ (line 17) and adding the tuple $\langle r_2, 4\rangle$ to $P_I$ (line 22). Then, function $\mathsf{disable}$ is invoked. The if condition in line 28 succeeds and the for loop selects rule $r_3$ to be deactivated. Therefore, in line 37 rule $r_3$ is also removed and the tuple $\langle r_3, 4\rangle$ is added to $P_I$. After reasoning at time points 2 and 3, rules $r_2$ and $r_3$ will be re-activated at time point 4 by adding them back to $P_A$ (line 10) and removing them from $P_I$ (line 11). In fact, it is only at this time point that these two rules can fire and produce some derivations.

The application of our method to Example 1 as shown before illustrates the benefit of our technique: The facts that some atoms were missing in one time point resulted in disabling two rules and for two time points reasoning was performed considering only $r_1$, and this can result in a better runtime.

## 5 Evaluation

We implemented a new reasoner in Golang which includes the optimization introduced in Laser and the technique proposed in this paper. The re-implementation was necessary since the pre-existing implementation of Laser was too prototypical to be extended. Throughout, we refer to the old Laser as "Laser1" and to the new implementation as "Laser2".

Below, we report the results of a number of experiments that we executed to illustrate the benefit introduced by our technique. The experiments can be grouped into four classes:

- *Ours vs. Laser1*: We compare the runtime vs. our old implementation;
- *Runtime single rule (best case)*: We study the runtime in the best case;
- *Runtime single rule (worst case)*: We study the runtime in the worst case;
- *Runtime multiple rules*: We observe the runtime with multiple rules.

**Inputs.** Although several benchmarks for stream processing exist (e.g., SR-Bench [39]) we are not aware of any that supports the operators in LARS and that can be used to stress the techniques introduced in this paper. In order to have full control on the experimental setting and to accurately measure the effects with the various configurations, we created, in a similar fashion as done in [4], a number of microbenchmarks that are specifically designed to evaluate our technique.

**Evaluation Setup.** We ran all the experiments on an iMac equipped with 8-core Intel(R) 2.60GHz CPU and 8GB of memory. We used Golang 1.13 to compile and run our system and Pypy 7.2.0 to run Laser1. To minimize the footprint of external effects (e.g., memory garbage collection, etc.) in our results, we run each experiment ten times over 300 time points and report the average result.

**Ours vs. Laser1.** Before we evaluate our proposal, we report some experiments where we compare the performance of Laser1 and Laser2 (the latter is executed without our proposed optimization). The motivation for doing so is to show that our new implementation is more performant than the old one, and this justifies its usage in the following experiments when we evaluate our technique.

In this set of experiments, we created a number of programs $P_n$ where $n \in \{10, 50, 100\}$ which contain a single rule of the form $\boxplus^n \Diamond p(X,Y) \to q(X,Y)$. Similarly, we have also created other programs $P'_n$ with the rule $\boxplus^n \Box p(X,Y) \to$

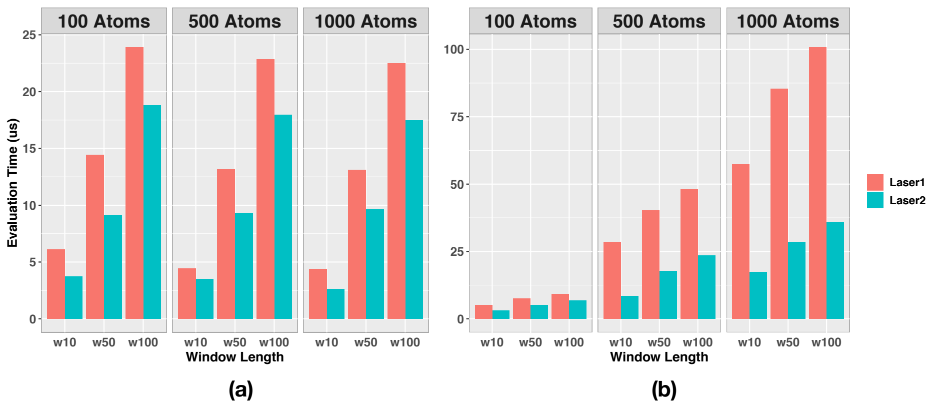

**Fig. 1.** Runtime of Laser1 and Laser2. In (a), $w_n$ refers to $P_n$. In (b), $w_n$ refers to $P'_n$.

$q(X, Y)$. Intuitively, $P_{10,50,100}$ test the performance of the reasoner with a rule that uses the $\Diamond$ operator while $P'_{10,50,100}$ does the same but with the $\Box$ operator.

In each experiment, we instruct the data generator to create, at each time point, the set of facts $\bigcup_{i:=1}^{m}\{p(a_i, a_i)\}$ where $m \in \{100, 500, 1000\}$. In this way, we can stress the system both varying the window size and the number of facts in input. The average reasoning runtime for processing one input fact with $P_{10,50,100}$ is reported in Figure 1a while the one with $P'_{10,50,100}$ is reported in Figure 1b.

From the figures, we can see that in both cases Laser2 outperforms Laser1. In addition, we can make some interesting observations about the operators $\Diamond$ and $\Box$. In Figure 1a, we can see that if the number of input atoms increases and the window size remains constant, then the average runtime remains relatively constant or even decreases. This behavior is due to the usage of the horizon time introduced in [4] which extends the validity of a formula for as many time points as the window size. Moreover, reasoning at each time point has a fixed cost that is amortized over the input facts. If there are more input facts, this cost becomes less prominent. This explains the slight decrease in the runtime when the input size increases.

The results in Figure 1b show a different behaviour. In this case, the validity of the body of the rule cannot be extended to the future. Consequently, the runtime increases both when the window size increases (since the reasoner has to check that the facts hold at more time points) and when the size of the stream increases. This shows that the evaluation of $\Box$ can be much more challenging than $\Diamond$. The increase of the runtime is observed with both implementations although with Laser2 it is less pronounced. The reason behind this difference is purely technical and due to the fact that the new implementation does not have the overhead introduced by the interpretation layer of Python.

**Runtime single rule (best case).** We now compare the runtime on a simple benchmark with and without activating our technique. We consider a series of programs which contain a single rule of the form $\boxplus^n \Diamond p(X, Y), \boxplus^n \Box q(X, Y) \rightarrow$

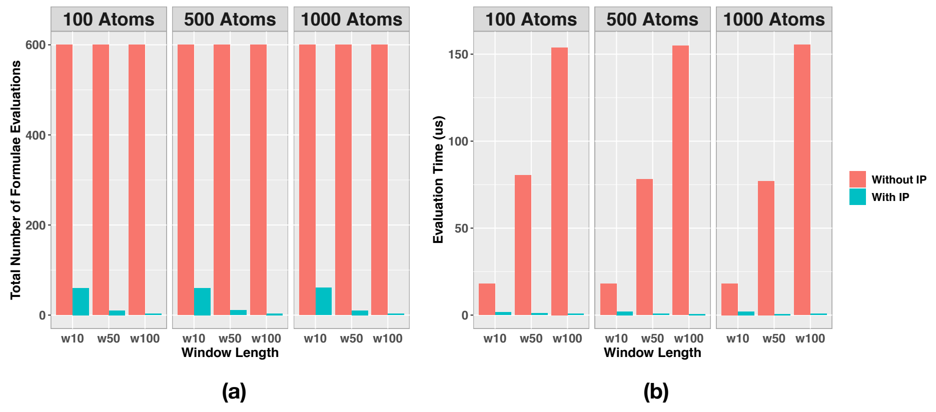

**Fig. 2.** Total number of evaluated formulae (a) and average runtime per input fact (b) in a best-case scenario.

$m(X, Y)$ where $n \in \{10, 50, 100\}$. We apply the programs on different streams. At time point $i$, the various streams contain the facts $\bigcup_{i:=1}^{m}\{p(a_i, a_i)\}$ if $i \bmod n = 0$ or $\bigcup_{i:=1}^{m}\{p(a_i, a_i), q(a_i, a_i)\}$ otherwise, where $m \in \{100, 500, 1000\}$. In essence, the idea is to use a stream where every $n$ time points there are no $q-$facts so that $\boxplus^n \Box q(X, Y)$ does not hold and consequently the rule is disabled. This scenario represents the best case for our method because without it the reasoner would need to evaluate the rule at each time point.

Figures 2a and 2b report the total number of formulae evaluations and the average runtime per input fact with different window and stream sizes. The results marked with "With IP" ("Without IP") use (don't use) our technique. The results show that with our approach the reasoner evaluates many fewer formulae (because the rule is disabled most of the time). Note that when our technique is enabled the number of evaluated formulae is non-zero. The reason is that every $n$ time points the counter for disabling the rule expires and the rule is re-added to the set of active rules. This event occurs less frequently if the window size is larger. This explains why the number of evaluated formulae decreases in Figure 2a.

As a consequence that some rules are disabled, the runtime decreases to the point it is barely visible in Figure 2b. It is worth to point out that in Figure 2b the runtime with our technique is almost constant while, without our technique, it increases with the window size (this behavior was observed also in Figure 1b). This comparison illustrates the effectiveness of our approach in disabling rules.

**Runtime single rule (worst case).** In the previous set of experiments, we evaluated our technique in a best-case scenario. We now present some experiments in a worst-case scenario. To simulate this case, we consider programs with the rule $\boxplus^{10} \Diamond p(X, Y), \boxplus^n \Box q(X, Y) \rightarrow m(X, Y)$ where $n$ (i.e., the window size) is very small. In particular, we considered $n \in \{1, 2, 3\}$. If $n = 1$ and the rule is disabled, then our approach immediately re-adds it in the next iteration since

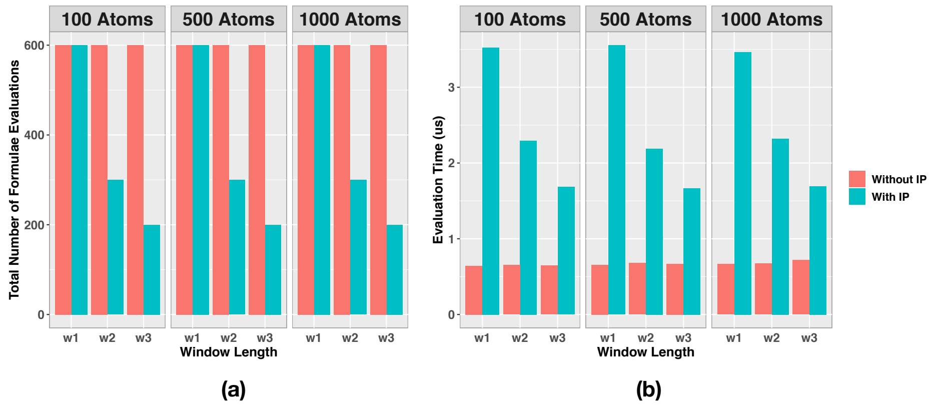

**Fig. 3.** Total number of evaluated formulae (a) and average runtime per input fact (b) in a worst-case scenario.

its invalidity has expired. For this reason, we can use this type of program to measure the overhead of our approach with an input where it is not effective. As input streams, we consider those that add the facts $\bigcup_{i:=1}^{m}\{p(a_i, a_i)\}$ at each time point where $m \in \{100, 500, 1000\}$. Note that since no fact with predicate $q$ appears in the stream, the rule will always try to disable it (unless it was already previously disabled) but in the next 1, 2, or 3 time points the rule will be re-activated.

Figures 3a and 3b report, similarly as before, the number of formulae evaluations and the runtime per input fact. From these results, we observe that when the window size is one (which is the worst scenario), the number of evaluations is the same as when our technique is disabled. However, the overhead incurred by our approach significantly increases the runtime. If the window size increases, then the performance improves because the overhead is less prominent.

Note that there is a simple optimization to overcome the problem observed in this experiment: When we populate $R$ in line 5 of Algorithm 1, we can consider only the formulae where the window size is sufficiently large (e.g., $x > 10$). In this way, we can restrict the application only to the cases whether the saving introduced by our technique outweighs the overhead.

**Runtime multiple rules.** We have shown that sometimes disabling a rule can have a cascading effect that leads to the disabling of more rules. To test the performance in this scenario, we consider a series of programs of the form

$$\boxplus^n \Diamond p(X,Y) \wedge \boxplus^n \Box q(X,Y) \rightarrow h_1(X,Y) \qquad (r_1)$$

$$h_1(X,Y) \rightarrow h_2(X,Y) \qquad (r_2)$$

$$\ldots \qquad (r_{3\ldots w-1})$$

$$h_{w-1}(X,Y) \rightarrow h_w(X,Y) \qquad (r_w)$$

where $n \in \{10, 50, 100\}$ and $w \in \{10, 20\}$. As input, we use a fixed stream which contains the facts $\bigcup_{i:=1}^{300}\{p(a_i, a_i)\}$ at each time point.

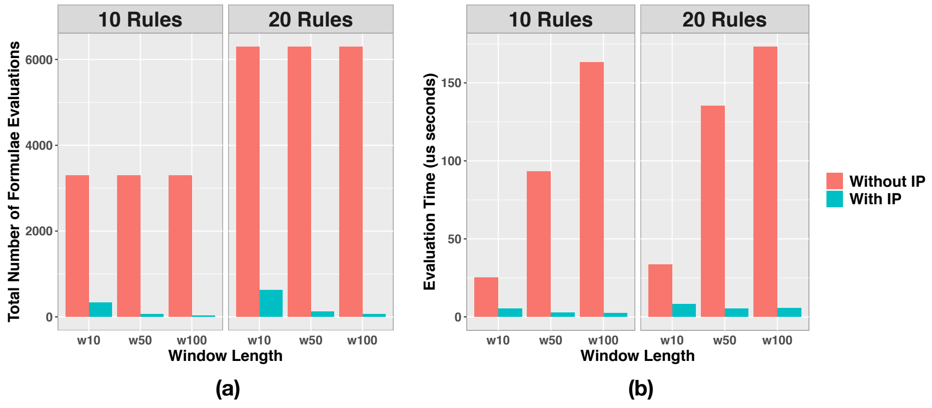

**Fig. 4.** Total number of evaluated formulae (a) and average runtime per input fact (b) with multiple rules.

Figures 4a and 4b report the total number of formulae evaluations and the average runtime per input fact respectively. We observe a similar trend as in the previous cases: Without our technique, at each time point the reasoner evaluates many more formulae and the runtime is significantly higher. With our technique, the average runtime drops and remains reasonably constant with different window sizes and number of rules (the slight increase is due to the overhead incurred by larger programs). The saving can be very high: In the best case (with $n = 100$ and $w = 20$), the runtime is 31 times faster. Although these numbers are obtained with artificially created datasets, they nevertheless indicate the effectiveness of our proposal in speeding up stream reasoning.

## 6 Related Work and Conclusion

**Related work.** The problem of stream reasoning in the context of the Semantic Web was first introduced by Della Valle et al. in [11, 12]. Since then, numerous works have been focused on different aspects of this problem and yearly workshops[1] have further fostered the creation of an active research community.

The surveys at [13, 21] provide a first overview of the various techniques. A few influential works have tackled this problem by extending SPARQL with stream operators [2, 3, 7–9]. Additionally, other stream reasoners either propose a custom processing model [20] or rely on (probabilistic) ASP [15, 25, 38] or on combinations of the two [22]. Finally, some works focus on improving the scalability [16, 26, 27] by distributing the computation on multiple machines or with incremental techniques [19]. Since these works support different semantics, it is challenging to compare them. Indeed, providing a fair and comprehensive

[1] The last one was in Apr.'19: `https://sr2019.on.liu.se/`

comparison of the various proposal remains an open problem, despite notable efforts in this direction [30, 33, 34].

The problem of stream reasoning has been studied also by the AI community. In [18], Koopmann proposes a new language to provide OBQA on temporal and probabilistic data. In [32], the authors investigate stream reasoning with Metric Temporal Logic (MTL) and later extend it with approximate reasoning [10]. In a similar setting, in [36], the authors consider stream reasoning in datalogMTL – an extension of Datalog with metric temporal operators. Finally, Ronca et al. [29] introduced the window validity problem, i.e., the problem of determining the minimum number of time points for which data must be kept in main memory to comply with the window sizes. None of these works address the problem of exploiting the impossibility of future derivations for improving the performance, as we do in this paper.

Finally, a research area that is closely related to stream reasoning is incremental reasoning [17, 24, 23, 28, 35, 37]. The major difference between incremental reasoning and stream reasoning is that the latter is characterized by the usage of windows functions to focus on the most recent data. Moreover, in a typical stream reasoning scenario data expires after a relatively short amount of time.

**Conclusion.** In this paper, we tackled the problem of providing efficient stream-based reasoning with (plain) LARS programs. In our previous work [4] we proposed a technique to reduce the number of redundant derivations by extending the time validity of formulae which will *hold* in the future. Here, we presented a technique to extend the time validity of formulae which will *not hold*. This is meant to target formulae where the previous technique is not effective.

Future work can be done in multiple directions. First, it is interesting to study whether more advanced techniques can determine a longer time validity (or invalidity) for formulae which are beyond plain LARS (e.g., nested windows). Moreover, a dynamic strategy can be designed to detect whether for some formulae a naïve recomputation is faster. Such a strategy could be used to mitigate the performance decrease observed in the worst-case scenario. Finally, our technique is triggered when no atoms with a certain predicate appear in the stream. It is possible that a more fine-grained technique, which considers facts rather than predicates, leads to improvements in more cases, but it is not trivial to implement it without introducing significant overhead.

Our experimental evaluation on artificially created microbenchmarks shows that the performance gain is significant. This makes our proposal a valuable addition to the portfolio of techniques for computing logic-based stream reasoning efficiently and at scale.

## References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.
2. Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a Unified Language for Event Processing and Stream Reasoning. In *Proceedings of WWW*, pages 635–644, 2011.

3. Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: a Continuous Query Language for RDF Data Streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.
4. Hamid R. Bazoobandi, Harald Beck, and Jacopo Urbani. Expressive Stream Reasoning with Laser. In *Proceedings of ISWC*, pages 87–103, 2017.
5. Harald Beck, Minh Dao-Tran, and Thomas Eiter. Lars: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence*, 261:16 – 70, 2018.
6. Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming*, 17(5-6):744–763, 2017.
7. Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL - Extending SPARQL to Process Data Streams. In *Proceedings of ESWC*, pages 448–462, 2008.
8. Pieter Bonte, Riccardo Tommasini, Filip De Turck, Femke Ongenae, and Emanuele Della Valle. C-Sprite: Efficient Hierarchical Reasoning for Rapid RDF Stream Processing. In *Proceedings of DEBS*, pages 103–114, 2019.
9. Jean-Paul Calbimonte, Óscar Corcho, and Alasdair J. G. Gray. Enabling Ontology-Based Access to Streaming Data Sources. In *Proceedings of ISWC*, pages 96–111, 2010.
10. Daniel de Leng and Fredrik Heintz. Approximate Stream Reasoning with Metric Temporal Logic under Uncertainty. In *Proceedings of AAAI*, pages 2760–2767, 2019.
11. Emanuele Della Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. A first step towards stream reasoning. In *Future Internet Symposium*, pages 72–81, 2008.
12. Emanuele Della Valle, Stefano Ceri, Frank Van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.
13. Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.
14. Thomas Eiter, Paul Ogris, and Konstantin Schekotihin. A Distributed Approach to LARS Stream Reasoning (System paper). *Theory and Practice of Logic Programming*, 19(5-6):974–989, 2019.
15. Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub. Answer Set Programming for Stream Reasoning. *CoRR*, abs/1301.1392, 2013.
16. Jesper Hoeksema and Spyros Kotoulas. High-performance Distributed Stream Reasoning Using S4. In *Ordring Workshop at ISWC*, 2011.
17. Pan Hu, Boris Motik, and Ian Horrocks. Optimised maintenance of datalog materialisations. In *Proceedings of AAAI*, pages 1871–1879, 2018.
18. Patrick Koopmann. Ontology-based query answering for probabilistic temporal data. In *Proceedings of AAAI*, pages 2903–2910, 2019.
19. Danh Le-Phuoc. Operator-aware Approach for Boosting Performance in RDF Stream Processing. *Journal of Web Semantics*, 42:38–54, 2017.
20. Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of ISWC*, pages 370–388, 2011.
21. Alessandro Margara, Jacopo Urbani, Frank Van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Journal of Web Semantics*, 25:24–44, 2014.

22. Alessandra Mileo, Ahmed Abdelrahman, Sean Policarpio, and Manfred Hauswirth. StreamRule: a Nonmonotonic Stream Reasoning System for the Semantic Web. In *International Conference on Web Reasoning and Rule Systems*, pages 247–252, 2013.
23. Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Incremental Update of Datalog Materialisation: The Backward/Forward Algorithm. In *Proceedings of AAAI*, pages 1560–1568, 2015.
24. Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of datalog materialisations revisited. *Artificial Intelligence*, 269:76–136, 2019.
25. Matthias Nickles and Alessandra Mileo. Web Stream Reasoning using Probabilistic Answer Set Programming. In *International Conference on Web Reasoning and Rule Systems*, pages 197–205, 2014.
26. Thu-Le Pham, Muhammad Intizar Ali, and Alessandra Mileo. Enhancing the scalability of expressive stream reasoning via input-driven parallelization. *Semantic Web*, 2019.
27. Xiangnan Ren and Curé et al. Strider R: Massive and distributed RDF graph stream reasoning. In *Proceedings of International Conference on Big Data*, pages 3358–3367, 2017.
28. Yuan Ren and Jeff Z Pan. Optimising Ontology Stream Reasoning with Truth Maintenance System. In *Proceedings of CIKM*, pages 831–836, 2011.
29. Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, and Ian Horrocks. The window validity problem in rule-based stream reasoning. In *Proceedings of KR*, pages 571–580, 2018.
30. Thomas Scharrenbach, Jacopo Urbani, Alessandro Margara, Emanuele Della Valle, and Abraham Bernstein. Seven Commandments for Benchmarking Semantic Flow Processing Systems. In *Proceedings of ESWC*, pages 305–319, 2013.
31. Jakob Suchan and et al. Out of Sight But Not Out of Mind: An Answer Set Programming Based Online Abduction Framework for Visual Sensemaking in Autonomous Driving. In *Proceedings of IJCAI*, pages 1879–1885, 2019.
32. Mattias Tiger and Fredrik Heintz. Stream Reasoning Using Temporal Logic and Predictive Probabilistic State Models. In *23rd International Symposium on Temporal Representation and Reasoning*, pages 196–205, 2016.
33. Riccardo Tommasini, Emanuele Della Valle, Marco Balduini, and Daniele Dell'Aglio. Heaven: A Framework for Systematic Comparative Research Approach for RSP Engines. In *Proceedings of ESWC*, pages 250–265, 2016.
34. Riccardo Tommasini, Emanuele Della Valle, Andrea Mauri, and Marco Brambilla. RSPLab: RDF Stream Processing Benchmarking Made Easy. In *Proceedings of ISWC*, pages 202–209, 2017.
35. Jacopo Urbani, Alessandro Margara, Ceriel Jacobs, Frank Van Harmelen, and Henri Bal. Dynamite: Parallel Materialization of Dynamic RDF Data. In *Proceedings of ISWC*, pages 657–672, 2013.
36. Przemyslaw Andrzej Walega, Mark Kaminski, and Bernardo Cuenca Grau. Reasoning over Streaming Data in Metric Temporal Datalog. In *Proceedings of AAAI*, pages 3092–3099, 2019.
37. Yifei Wang and Jie Luo. An Incremental Reasoning Algorithm for Large Scale Knowledge Graph. In *International Conference on Knowledge Science, Engineering and Management*, pages 503–513, 2018.
38. Ying Zhang, Pham Duc, Oscar Corcho, and Jean-Paul Calbimonte. SRBench: a Streaming RDF/SPARQL Benchmark. *Proceedings of ISWC*, pages 641–657, 2012.
39. Ying Zhang, P. Minh Duc, O. Corcho, and J. P. Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In *Proceedings of ISWC*, pages 641–657, 2012.