Multi-context incremental reasoning over data streams

Wafaa Mebrek, Amel Bouzeghoub

Multi-Context Incremental Reasoning over Data Streams

Wafaa Mebrek[1,2] and Amel Bouzeghoub[1]
[1]Samovar, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France
[2]LabRI-SBA Laboratory, Ecole Supérieure en Informatique, Sidi Bel Abbes, Algeria
{Wafaa.Mebrek, Amel.Bouzeghoub}@telecom-sudparis.eu, w.mebrek@esi-sba.dz

***Abstract*—Data streams are generally generated on-the-fly and contain valuable knowledge to support the decision process in many use cases, such as smart cities or traffic monitoring. To this end, reasoning over streaming data suggests a reconsideration of the concept of a solution to a problem, and how to obtain it. As data in streams change continuously, all the conclusions based on expired data must be retracted, and additional or new information might also arrive and output additional derivations. Multiple works have investigated the use of Answer Set Programming (ASP) for streaming data. However, they generally do not provide window mechanisms, which are key factors in stream processing. Ticker is one of the recent solutions provided to tackle this issue. It is based on LARS, a Logic-based Framework for Analytic Reasoning over Streams, and has two reasoning strategies. One relies on Clingo for repeated solving, and the other uses a Truth Maintenance System to perform model updates. Despite the significant achievements that have been made, there is still room for improvement, especially when considering real-life applications. In this paper, we propose a new stream reasoning engine based on Ticker that overcomes its limitations. The proposed framework has been implemented and evaluated with a real-world benchmark, and a use-case scenario implementation shows promising results.**

***Index Terms*—Incremental Reasoning, Multi-context,Truth Maintenance System.**

I. Introduction

Today's world can be seen as an ocean composed of many different sources of data streams holding valuable knowledge and insights that can be leveraged to support decision-making in a wide range of use cases, such as smart cities, traffic monitoring, social media analysis and the Internet of Things (IoT). Stream reasoning is recognized as one of the most significant research areas supporting the aforementioned use cases by providing meaning to multiple, heterogeneous, massive, and inevitably noisy data streams to support the decision-making process of a very large number of simultaneous users [1].

Reasoning over streaming data suggests reconsidering the concept of a solution to a problem and how to obtain it. In contrast to static data, streams are constantly changing, as are their evaluations: conclusions based on out-of-date data need to be withdrawn, additional information may come in, giving rise to new derivations or even challenging previous ones, and so forth. How to manage dynamic environments in which incremental reasoning over heterogeneous knowledge sources with constantly arriving data streams remains largely unexplored in the literature. The main recent contribution is the Logic-based framework for Analytic Reasoning over Streams (LARS) [2], which is an extension of Answer Set Programming (ASP). It introduces window operators and time modalities, allowing for declarative specification of intricate decision problems involving streaming data. Ticker [3] and LASER [4] are the main implementations of LARS. Ticker, is a prototypical engine for well-defined logical reasoning over streaming data. It comes with two reasoning strategies, the ASP solver Clingo [5] and JTMS [6] [3], a Justification-based Truth Maintenance System [7], to maintain the model continuously. Despite the fact that this is a fully incremental stream reasoning system, we have identified a range of limitations : (i) neither Ticker nor JTMS can cope with contradictions; (ii) for each rule, it is mandatory to indicate the facts in the LARS file, which is in contradiction to a real-life application where facts occur on-the-fly. This makes it difficult for the incremental process to run correctly; (iii) variables in the scope of a window atom are grounded upstream in the pre-grounding phase by some standard atoms. As a consequence, some executions do not yield any results since the variables are replaced by the number of facts; (iv) JTMS works in only one context at a time and becomes increasingly costly as the volume of data grows. It is also challenging to determine whether a datum is derivable or not from a particular set of enabled assumptions. LASER is a system that updates models efficiently by applying a semi-naive evaluation of Datalog programs based on LARS instead of ASP. It restricts programs to positive and stratified ASP rules, limiting its expressiveness compared to Ticker. As a result, Ticker is currently the leading ASP-based stream reasoning engine that supports window operators and provides an incremental model update mechanism. However, there is still room for improvement in order to overcome some of the aforementioned shortcomings. In that respect, we propose in this paper LEA, a novel incrementaL strEam reAsoning engine based on Ticker.

A. Motivating Example

This example is extracted from CityBench [8], a comprehensive benchmarking suite to evaluate RDF Stream Reasoning engines within smart cities applications. The scenario is dedicated to public safety during a large event. The city is hosting a large cultural event at a popular venue in the city center that is expected to attract a large number of visitors. To manage the event and ensure public safety, the organizers have put several rules in place. During the event,

public safety is a top priority, and the city could implement increased security measures, such as increased police presence and surveillance cameras. Additionally, the city may need to restrict access to certain areas during the event to prevent overcrowding and congested roads and to maintain a safe environment, leading to limited parking capacity. Moreover, if a cultural event occurs in an area, and the roads are congested with activities, this will increase pollution. When the pollution level exceeds 50, the air quality is low, and the area is declared a high-emission zone. The authorities will either prohibit cars from entering the zone or limit the number of vehicles allowed there. This decision is expected to mitigate the negative effects of high pollution levels on public health and the environment. When it came to expressing these rules in LARS, we were confronted with several difficulties. For instance, to express the average speed V in a specific location L, we want to use the atom $averageSpeed(L, V)$. Unfortunately, using more than one argument in LARS is actually impossible. Moreover, the contradiction cannot be expressed with LARS. We cannot express, for instance, that both $freeRoadActivity$ and $congestedRoadActivity$ facts cannot occur simultaneously in the same location L and are, therefore, contradictory. Furthermore, we are constrained to indicate all the associated facts for each rule. But this is obviously impossible in a real-time application where the events arrive progressively.

Based on the aforementioned observations in our real scenario, an extended LARS shall be designed to meet the following requirements: (R1) Increase the expressiveness of logic rules to support multi-argument atoms and string types; (R2) Cope with conflicting rules; (R3) Processing the facts on the fly, rather than hard-coded in the rules file at initialization; (R4) Multi-context should be addressed.

B. Contributions and Novelty

In a nutshell, the novelty and the originality of our approach lies in the following main contributions:

- Extended-LARS: We propose an extended formula of LARS to express contradictory rules (denoted by the symbol "$\perp$") in a time window. Table I provides an example of program of this scenario expressed with the extended LARS.
- On-the-fly Grounding of plain LARS Program: We propose LEA, (short name for incrementaL strEam reAsoning), a new grounding technique that addresses the aforementioned shortcomings.
- Extended-ATMS Algorithm: We propose to use ATMS (Assumption-based Truth-Maintenance System) instead of JTMS. Indeed, JTMS is not applicable for programs with odd loops i.e., an odd number of negations, and is also far less efficient than modern ASP solving techniques; in particular when a model needs to be computed from scratch. It is also limited to ground programs. Consequently, providing other incremental model update techniques based on changing programs would be not only of interest for stream reasoning with LARS but also for ASP without an explicit timeline. Furthermore, we optimize the propagation function of the original version of ATMS. We have also added negative assumptions that are not supported by ATMS.
- We conduct extensive experiments on real datasets and compare our framework with Tiker and Laser. The results demonstrate promising results in terms of expressivity while maintaining fairly similar performances

The remainder of this paper is organized as follows: Section II introduces the formal extended-LARS definitions. Section III outlines the proposed framework, Section IV discusses the literature review, and Section V details the experimental evaluation. Finally, Section VI concludes the paper.

II. Extended-LARS

Adding Contradiction. JTMS cannot deal with inconsistency. Authors of Ticker, chose not to address the issue of this procedure. We introduce hereafter the contradiction concept and define the notion of conflicting rules. We use plain LARS programs syntax proposed in [3].

A ground plain LARS program P is a set of rules of the form $\alpha \leftarrow \beta_1, ...\beta_j, \text{not } \beta_{j+1}, ..., \text{not } \beta_n$,

where the head α is of form a or $@_t a, a \in A$, a set of atoms and $\beta_1....\beta_n$ is the body.

Definition 1: **Formulas:** The set of formulas F in LARS is defined by the following grammar to which we add the contradiction:

α : : = a | $\neg\ \alpha$ | $\alpha \vee \alpha$ | $\alpha \wedge \alpha$ | $\alpha \rightarrow \alpha$ | $\diamond\ \alpha$ | $\Box\ \alpha$ | $@_t\ \alpha$ | $\boxplus_i\ \alpha$ | $\perp$

where $\boxplus_i$ is a type of window operator, a $\in$ A is an atom and t $\in \mathbb{N}$.

If the contradiction formula, denoted $\perp$, is an atom in A, it can be treated as logically equivalent to any inconsistent formula. So if $a \vdash \perp$ (where $\vdash$ is the classical consequence relation), then a is inconsistent.

Definition 2: **Structure:** A structure M is a tuple $M(T, v, W, B)$ where $S = (T, v)$ is a stream, W a set of window functions, and B a set of facts.

Definition 3: **Entailment:** When formulas hold in a structure, an entailment relation $\models$ is defined between (M, S, t) and formulas as follows: Let $a \in A$, $w \in W$ and $\alpha, \beta \in F_g$ be the set of ground formulas where each term is ground. Then,

M,S,t $\models$ a iff a $\in v(t)$ or a $\in$ B,
M,S,t $\models \neg\ \alpha$ iff M,S,t $\not\models \alpha$,
M,S,t $\models \alpha \wedge \beta$ iff M,S,t $\models \alpha$ and M,S,t $\models \beta$,
M,S,t $\models \alpha \vee \beta$ iff M,S,t $\models \alpha$ or M,S,t $\models \beta$,
M,S,t $\models \alpha \rightarrow \beta$ iff M,S,t $\not\models \alpha$ or M,S,t $\models \beta$,
M,S,t $\models \diamond\ \alpha$ iff M,S,t' for Some t' $\in$ T,
M,S,t $\models \Box\ \alpha$ iff M,S,t' $\models \alpha$ for all t' $\in$ T,
M,S,t $\models$ @t'α iff M,S,t' $\models \alpha$ for all t' $\in$ T,
M,S,t $\models \boxplus^w\ \alpha$ iff M,S',t $\models \alpha$ where S' is a substream of S,
M,S,t $\models \perp$ iff M,S,t $\models \alpha$ and M,S,t $\models \neg\ \alpha$.

If M,S,t $\models \alpha$ holds, we say that (M,S,t) entails α. Moreover, M satisfies α at time t. Let $I = (T, v)$ be a stream such that $S \subseteq I$. If at every time point in T, all atoms that occur in I but not in S have intensional predicates, then we call I an

TABLE I
EXTENDED LARS PROGRAM FOR THE MOTIVATING EXAMPLE

$r_1 : @T_{limitedParking(L)} \leftarrow \boxplus^{60}_{@T}, culturalEvent(L), parkingCapacity(L, C), C \leq 20, area(L)$
$r_2 : @T_{congestionRoad(L)} \leftarrow \boxplus^{60}_{@T}, averageSpeed(L, V), V \leq 20, area(L)$
$r_3 : @T_{congestedRoadActivity(L)} \leftarrow \boxplus^{60}_{@T} limitedParking(L), \boxplus^{60}_{@T} congestionRoad(L)$
$r_4 : @T_{freeRoadActivity(L)} \leftarrow closedRoad(L), area(L)$
$r_5 : \bot \leftarrow @T_{freeRoadActivity(L)}, @T_{congestedRoadActivity(L)}$
$r_6 : @T_{poorAirQuality(L)} \leftarrow congestedRoadActivity(L), pollution(P), P \geq 50$
$r_7 : @T_{limitedAccessArea(L)} \leftarrow congestedRoadActivity(L)$
$r_8 : @T_{increaseSecurity(L)} \leftarrow \boxplus^{60}_{@T} congestedRoadActivity(L), limitedArea(L), crowdedEvent(L), highEmissionZone(P)$

interpretation stream for S and a structure $M = <T, v, W, B>$ an interpretation (for S).

Conflicting rules may occur when two (or more) rules can be simultaneously applied but their conclusions are in conflict. In other words, both outputs cannot be correct with respect to the intended interpretation I. Let us consider two rules r and $r\prime$ of the form $\psi \rightarrow c$ and $\psi\prime \rightarrow c\prime$ respectively. We define conflicting rules as follows:

Definition 4: **Conflicting Rules:** r and $r\prime$ are conflicting if there exists a state described by formula ϕ, such that simultaneously $\phi \models \psi$ and $\phi \models \psi\prime$ but $\not\models c \wedge c\prime$ under the assumed interpretation I (i.e. c and $c\prime$ cannot be simultaneously true, for example $c = \neg c\prime$ which cannot be true under any interpretation).

Example 1: In the motivating example presented above in Table I, r_3 and r_4 are conflicting since $@T_{congestedRoadActivity(L)} = \neg @T_{freeRoadActivity(L)}$

III. MULTI-CONTEXTS INCREMENTAL REASONING FOR PLAIN LARS PROGRAM

Through this section, we explain in detail the two key components of our proposed approach, LEA. The first component involves the generation of incremental rules based on incoming facts from a signal set, referred to as *On-the-fly Grounding*. The second component is an extension of ATMS implementation. By separating the On-the-fly Grounding process from the ATMS, a modular and flexible approach to knowledge base updates is achieved.

A. On-the-fly Grounding of plain LARS Program

On-the-fly Grounding operates independently from ATMS and forwards its results to ATMS for further processing and integration into the overall knowledge base. Algorithm 1 describes the mechanism for dynamically processing and annotating LARS rules based on the current tick time, tick count, and signal set, enabling efficient and accurate reasoning in the context of temporal logic-based systems.

It takes as input the current tick time t, the tick count c, and a signal set Sig with at most one input signal, which is empty if (t, c) is a time increment. In Lines 1 and 2, a rule body composed of a set of atoms is defined, and a set of valid rules is initialized. F in line 3 is a set of annotated rules with tick t and count c. These rules expire neither based on time nor count, hence the duration annotation is (∞, ∞).

In Line 4, auxiliary facts with time-pinned atom $a@(x, t)$ or counts $a_{\#}(x, t, c)$ are added to a fresh set F due to the translation of a LARS program P to an ASP program $\hat{P}$ (see Algorithm 1 in [3]). Every time a signal $a(x)$ is found, the *groundingRulesWithVariables* function (Line 5) is called to identify the rules in the LARS program P that match the signal $a(x)$ in the signal set Sig. This function (detailed below in Algorithm 2) takes as input the set of rules that match the received facts and the signal $a(x)$, and returns a set of rules that match the signal $a(x)$. These matching rules are grounded using the current tick time t and the received facts. This grounding process replaces variables in the rule with their corresponding values from the received facts and replaces the time variable with its current value. The output of this step is a set of grounded rules that are specific to the current tick time t and received facts. The *pinRules* function is used to pin the grounded rules generated in Line 5, with the corresponding tick count c. The pinned rules are added to the set of incremental rules R in Line 6. By pinning the rules, Algorithm 1 ensures that the rules remain active and are not removed from the program until their expiration time, which is determined by the tick count c. The set of incremental rules R (Line 8) represents the set of rules that need to be added to the program to accommodate the changes in the current state. This step is crucial for maintaining the program's consistency with respect to the current state and enables efficient processing of the LARS program. The algorithm also creates the set of window rules Q (Line 9) for each predicate in the LARS program P. These window rules are used to capture the current state of the system and are updated as new facts arrive. Note that the window rules have a duration of infinity, meaning that they remain valid until they are explicitly retracted. After processing all the signals, the algorithm generates the incremental rules for every rule in the LARS program P. First, a base rule $\hat{r}$ for each rule r is created (Line 11). Then, for every encoding e in the body of r, a set of incremental window rules I is created by *incrementalWindowRules* function (Line 12). The aim of generating these rules is to improve the accuracy and efficiency of the LARS program by providing incremental updates to the existing rules. This allows the system to react to new information and make adjustments in real time, which is particularly useful in dynamic environments where the input data is constantly changing. The last step is to return the incremental rules annotated with duration until expiration in Line 15. These rules serve as inputs in the ATMS algorithm 7, that construct the justification graph.

Algorithm 2 plays a crucial role in identifying and extracting

Algorithm 1 On-the-fly Grounding (Sig)

Input: Tick time t, tick count c, signal set Sig with at most one input signal, which is empty iff (t, c) is a time increment, P : the LARS program.
Output: Pinned incremental rules annotated with duration until expiration.

1: $\beta = atom_1, atom_2, \ldots, atom_n$ ▷ The body of the rule
2: $R := \emptyset$ ▷ The valid rule set at t
3: F := $\langle(\infty, \infty), \text{tick}(t, c) \leftarrow\rangle$
4: **for all** $a(x) \in Sig$ **do**
$F := F \cup \langle(\infty, \infty), a@(x, t) \leftarrow\rangle, \langle(\infty, \infty), a_{\#}(x, t, c) \leftarrow\rangle$
5: groundedRules := groundingRulesWithVariables (P, a(x), β)
6: pinnedRules := pinRules(groundedRules, c) ▷ The annotated rules with duration
7: **end for**
8: $R := R \cup$ pinnedRules
9: $Q := \langle(1, \infty), a(x) \leftarrow a@(x, t)\rangle, \langle(1, \infty), a@(x, t) \leftarrow a(x)\rangle$
10: **for all** $r \in P$ **do**
11: $\hat{r} :=$ baseRule(r)
12: $I := \bigcup_{e \in B(r)}$ incrementalWindowRules(e, t, c)
13: $R := R \cup I \cup \langle(\infty, \infty), \hat{r}\rangle$
14: **end for**
15: **return** $F \cup Q \cup R$ ▷ The resulted incremental rules

Algorithm 2 groundingRulesWithVariables(P, a(x), β)

Input: Pre-grounded LARS program P, the signal $a(x)$ and the set of atoms β
Output: A set of ASP rules K

1: $K := \emptyset$
2: **for all** $r \in P$ **do**
3: **if** $\beta \neq \emptyset$ and $a \in \beta$ **then**
$r' := replaceArgWithX(x)$
$K := K \cup r'$
4: **end if**
5: **end for**
6: **return** K

ASP rules from a pre-grounded LARS program P. This algorithm takes the pre-grounded LARS program P as input and aims to find rules that contain a specific variable $a(x)$. It initializes Line 1, an empty set K, to store the identified ASP rules and iterates (Lines 2-5) through each rule in P. Line 3 checks whether the set of atoms defined by β is not empty, and the signal a is present. In this case, it creates a new rule r' by replacing the argument of the atom a with the variable x and adds this modified rule to the set K of identified ASP rules.

Example 2:

Let us consider the following LARS rule:

$$r = @T_{congestionRoad(L)} \leftarrow \boxplus^{30}_{@T} averageSpeed(L, V), \\ V \leq 20, area(L).$$

where the predicate $averageSpeed(L, V)$ serves as a guard that always remains valid. After translation of the LARS program to ASP rules using the proposed Algorithm 1, we get the following rule:

$$\hat{r} = congestionRoad(L)\_@(T) \leftarrow \omega_e averageSpeed(L, V, T), \\ \text{Leq}(V, 20), area(L).$$

where $\omega_e = \boxplus^{30}_{@T} averageSpeed(L, V)$. To ground the rule, we replace the variable V in the window atom $\boxplus^{30}_{@T} averageSpeed(L, V)$ with the value from the signal input. For instance, when the program receives the fact $averageSpeed(\text{"}StreetA\text{"}, 18)$, the resulted incremental rule is as follows :

$$congestionRoad(\text{"}StreetA\text{"})_@(T) \leftarrow \\ \omega_e averageSpeed(\text{"}StreetA\text{"}, 18), \\ \text{Leq}(18, 20), area(\text{"}StreetA\text{"}).$$

Unlike Tiker, we propose not to include the $averageSpeed$ facts in the LARS file but instead receive it gradually from a stream as a signal.

This On-the-fly Grounding of variables in @-atoms avoids the need to declare a fact in LARS rule file for each atom with a dynamic argument and provides more efficient and flexible incremental grounding. In the end, the resulting incremental rule has an infinite duration like the base rule. The reason why we assign it is that it does not need to expire. In other words, the base rule is a static rule that does not depend on the stream of input data and is not subject to any changes over time. Therefore, it can remain valid indefinitely. This approach simplifies the implementation of the LARS program and improves its performance by avoiding unnecessary overheads associated with rule expiration and re-evaluation.

B. Extended ATMS algorithms

The Assumption-based Truth Maintenance System (ATMS) algorithm stands out as a type of incremental Truth Maintenance System (TMS) algorithm and plays a crucial role in maintaining consistency, tracking dependencies, and managing logical inferences within the knowledge base.

Its core process involves several key algorithms, including UPDATE (cf. Algorithm 3), a primary function that updates the labels of nodes incrementally by appending justifications, thereby ensuring the local correctness of these labels; WEAVE (cf. Algorithm 6), which manages the weaving of justifications and dependencies; PROPAGATE (cf. Algorithm 5), which propagates changes and updates to affected elements; and NOGOOD (cf. Algorithm 4), which handles and detects conflicts or contradictions. Together, these algorithms form the foundation of ATMS and enable efficient reasoning and maintenance of knowledge within a multi-context environment as in Table I. Furthermore, the ATMS follows a systematic approach where it continually propagates alterations until the labels reach a state of global correctness and initializes the labeling process by tagging assumptions with the corresponding environment that encloses them. Consequently, all the remaining nodes are generated devoid of any labels, i.e., they are created with empty labels.

In ATMS, the UPDATE algorithm ensures that a node contains a contradiction in order to guarantee nogoods detection from the outset (cf. Algorithm 3). It starts by detecting the nogoods operates (Lines 1-5) and eliminates from all nogoods along with environments that are subsumed by others within L. This step ensures that the set L is devoid of redundant or unnecessary elements and simplifying the remaining computation (Algorithm 3, Lines 6-8). For every justification J in which n is mentioned as an antecedent, the PROPAGATE Algorithm 5 is called to incrementally change the

Algorithm 3 UPDATE (L, n)

1: **if** $n = \perp$ **then**
2: **for** each E in L **do**
3: $NOGOOD(E)$ ▷ Removal of Nogoods and Subsumed Environments
4: **end for**
5: **end if**
6: Remove all environments from L that are a subset of any environment in the label of n. ▷ Update n's label, ensuring minimality
7: Remove all environments from the label of n that are a subset of any element of L.
8: Include all remaining environments of L in the label of n.
9: Ensure negation.
10: **for** each $j \in J$ and antecedant(n) **do** ▷ The variable J is declared in a scope external to the UPDATE method
11: $PROPAGATE(J, n, L)$ ▷ Propagate the incremental change to n's label to its consequences
12: **end for**
13: Remove subsumed and inconsistent environments from L.
14: Remove from L all environments no longer in n's label.

Algorithm 4 NOGOOD (E)

1: Mark E as noogood
2: **for** each node n **do** ▷ Iterate over all nodes in the dependency network.
3: **for** each environment E' in node n's label **do**
4: **if** E' is a superset of E **then**
5: Remove environment E from node n's label.
6: **end if**
7: **end for**
8: **end for**

label of n with respect to its consequences. Then, all subsumed and inconsistent environments are removed from L (Lines 10-14).

The NOGOOD Algorithm 4 is invoked whenever a newly created environment E is marked as nogood (Line 1). It then traverses every node n within the dependency network and eliminates all nogood environments from all labels as shown in Lines 2-8. This ensures that any conflicting or invalid environments are eliminated, promoting consistency and accuracy in the reasoning process.

The PROPAGATE Algorithm 5 consists of two main steps : The first step (Line 1) involves computing the incremental label update using the WEAVE Algorithm (cf. Algorithm 6). In the second step (Line 2), the UPDATE Algorithm is invoked to propagate the changes further (cf. Algorithm 3). The proposed PROPAGATE Algorithm in [9] and [7] is inefficient because it recomputes node labels repeatedly in step 2. We propose a more efficient approach in which only incremental changes to node labels are propagated. This improved algorithm is provided in our extended ATMS in Algorithm 5.

The WEAVE algorithm, depicted in Algorithm 6, performs a crucial step in the incremental label update process. It operates on an antecedent node a, a set of current environments I, and a set of antecedent nodes X (Lines 1-4). It incrementally computes a tentative label (denoted by L) by combining the environments from I and the label of each antecedent node h (Line 5). During the label construction, conflicts between environments are checked to ensure consistency (Line 6). The resulting label I' contains no known inconsistency. Finally, the recursive call of the WEAVE Algorithm 6 (Line 10) allows for further processing and refinement of the label, taking into account additional antecedent nodes and their associated environments.

Finally, ADDJUSTIFICATION algorithm (Algorithm 7) is a crucial component that bridges the gap between ATMS and the reasoner, with a particular emphasis on the on-the-fly grounding

Algorithm 5 PROPAGATE (J, n, l)

1: $L \leftarrow$ WEAVE$(a, I, \{X1, \ldots, Xk\})$ ▷ Weave operation to determine new labels
2: UPDATE(L, n) ▷ Add the new potential label environments L to node n

Algorithm 6 WEAVE (a, I, X)

1: Iterate over antecedent nodes $h \neq a$
2: **if** $X = \emptyset$ **then**
3: **return** I
4: **end if**
5: Incrementally build the incremental label L
6: Check for conflicts between environments during the construction.
7: Let $I' = \bigcup_{e_i \in I} \bigcup_{f_j \in h.\text{label}} \{e_i \cup f_j\}$
8: Ensure that I' is minimal and contains no known inconsistencies.
9: Clean(I') ▷ Eliminate duplicates, invalidations, and any environment that is subsumed by any other environment from I'
10: **return** $WEAVE(a, I', X)$

phase sketched in Algorithm 1. It takes as input incremental rules, denoted as R, which are the output of Algortihm 1, and executes a series of steps to facilitate the integration of the rule within the ATMS framework. The key step occurs in Line 2, where the convertToJustification function is employed to convert the incremental rule R into a justification, denoted by J. This conversion enables the subsequent utilization of the rule within the ATMS graph as shown in Figure 1. Finally, in Line 3, the PROPAGATE Algorithm 5 is called.

Algorithm 7 ADDJUSTIFICATION (R)

$R : = \alpha \leftarrow \beta_1, \ldots \beta_j, \text{not } \beta_{j+1}, \ldots, \text{not } \beta_n.$
1: $L := \emptyset$
2: $J := convertTojustification(R)$
3: $PROPAGATE(J, n, L)$

Adding Negation. Based on a general labeling algorithm for ATMS [9], we propose an extension to ensure that the ATMS graph generation properly handles negated literals, allowing for more accurate reasoning and conflict detection.Thus, some modifications have been introduced compared to the original version. Instead of searching within the antecedents of the nogood node justifications, the algorithm now performs a search for the assumption's negation, as indicated in Algorithm 3.

Example 3: Figure 1 shows the ATMS graph resulting from the motivating example described in Table I. The ATMS graph represents the dependencies and justifications between the assumptions made by these rules. In this example, the ATMS graph includes nodes representing the possible assumptions, for example: $congestedRoadActivity(L)$, $averageSpeed(L, V)$, $limitedAccessArea(L)$, and justifications : the rule r_1 justifies the assumption $limitedParking(L)$ based on the assumptions $culturalEvent(L)$, $parkingCapacity(L, C)$, $C \leq 20$, and $area(L)$. The nodes marked in blue in the ATMS represent different contexts that the ATMS manages and can use to easily switch between contexts. This feature enables it to handle multi-context scenarios as : $Parking(L, C)$, $leq(C, 10)$ and $Area(L)$. These context switches are not costly because they simply require choosing a different second argument. In addition, the LEA's inference engine can now work in multiple contexts at once. All black nodes represent justifications,

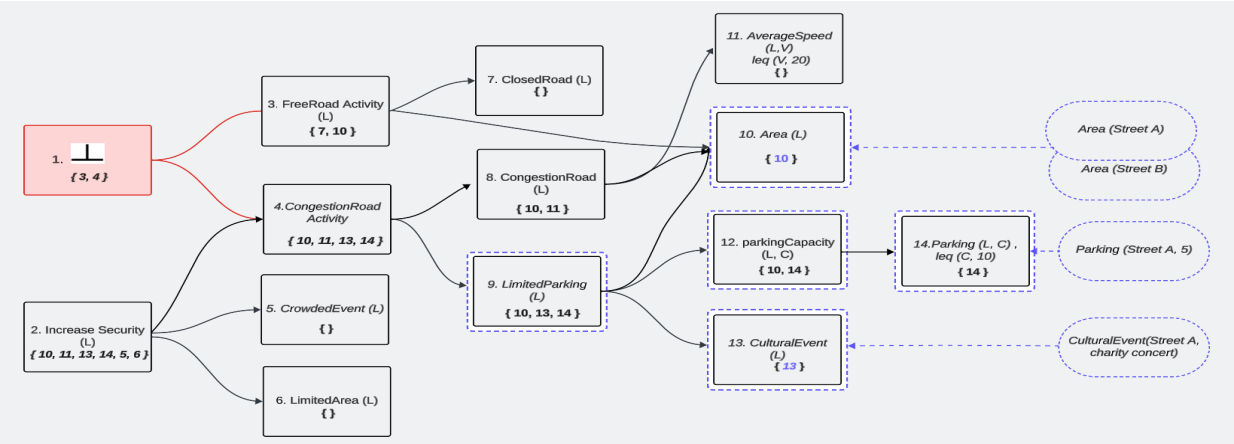

Fig. 1. ATMS graph modeling

validated nodes are in blue and the numbers enclosed in curly braces $\{\}$ represent the labels of these nodes. By analyzing the ATMS graph, it becomes possible to track the logical consequences of the assumptions and to determine the overall truth values and conflicts within the rules system to detect and handle contradictory rules : $CongestionRoadActivity(L)$ and $FreeRoadActivity(L)$ as depicted in r_3 and r_4 (red node).

IV. Related Work

Stream reasoning has been widely studied in the last few years [10] and many advanced stream processing solutions were developed in the context of Data Stream Management Systems (DSMSs) and Complex Event Processors (CEPs). However, to date, there are no standards that make the comparison between these approaches. Proposals based on ASPs are of particular interest, whereas those related to DSMSs and CEPs are beyond the scope of this paper.

Several works have explored reasoning over streams using the ASP solver Clingo, addressing the challenges of data and program changes. Incremental grounding and solving techniques have been introduced in incremental ASP. Ideas from incremental ASP, reactive ASP, and time-decaying logic programs have been continuously improved and are now integrated into the current version 5 of Clingo. The multi-shot solving capabilities of Clingo [11] [12] to evaluate changing programs were presented earlier, providing additional control for grounding and solving through external script-accessible parameters. These mechanisms can be used, for example, to simulate the progress of time and encode window operators. However, while Clingo's multi-shot features target incremental and reactive control of the ASP-solving process, they do not offer explicit streaming semantics or operators. The LARS framework is the first formal semantics for stream reasoning and extends ASP. It incorporates window operators to capture data snapshots with time, tuple, or partition, and temporal operators for evaluation at specific time points or intervals in a stream. In contrast to ASP, with LARS , it is possible to express time as time points or intervals within the rules. Ticker and Laser have implemented fragments of LARS programs, both based on plain LARS rules. Ticker focuses on negation and offers two evaluation modes : a static ASP encoding that uses Clingo for repeated solving, and an incremental ASP encoding that performs model updates using truth maintenance techniques. On the other hand, Laser focuses on positive and stratified programs with sliding windows, extending the semi-naive evaluation of Datalog. Both Ticker and Laser are capable of updating previously computed models when data streams change. However, they differ in that Ticker is based on LARS and not ASP, and Laser only accepts the positive fragment of LARS and stratified programs with sliding windows. Ticker supports full negation with sliding time/tuple-based windows.

V. Experimental Evaluation

In this section, we describe the experiments conducted to demonstrate the effectiveness of the proposed approach using On-the-fly Grounding and ATMS. The experiments aim to answer the following four research questions:

– **RQ1.** Is there a reasoning engine among LEA, Ticker, and Laser that performs significantly better, or do they perform comparably?

– **RQ2.** Which reasoning engine offers the best results in terms of quality?

– **RQ3.** What impacts the incremental evaluation of rules or the grounding phase?

– **RQ4.** How do the different parameter settings influence the performance of reasoning engines?

Datasets. In order to compare the performance of our framework, we used two implemented scenarios in Ticker, a *Content Retrieval* that can accommodate multiple models and utilizes recursive computation instead of a linear sequence of instructions. It is designed for a network where items can be cached and requested at any node, and *Caching Strategy* where the program enables the selection of one of four replacement strategies (fifo, lfu, lru, or random) to manage the removal of video chunks from a local cache. Due to space constraints, we do not describe these scenario. More details can be found in [3].We have modified the scenarios for our tests and added

contradictory rules and operations on the arguments to measure the quality of each implementation.

Experimental Setup. Our focus in this study was to investigate the potential algorithmic benefits of incremental reasoning using our proposed solution and the incremental reasoner mode of Ticker. Therefore, we did not compare our approach with the push-based ASP reasoner, but we rather focused solely on incremental evaluation. The LARS program representation does not include a notion of clock time, so we did not fix a clock time and test how many atoms can be processed per time point. Instead, we processed each tick as fast as possible, which allowed us to compare the maximal performance of the incremental mode without introducing additional overhead, such as input/output handling. On the other hand, at the time of publishing Laser [4], the incremental reasoning mode of Ticker was not available for evaluation. Therefore, the authors only tested their approach using the Clingo reasoning mode.

Evaluation. We performed two types of evaluation modes for the scenarios and setup. The first mode involved fixing the number of time points and gradually increasing the window size n, while the second mode did the opposite. In each mode, we measured three parameters: (i) the time required to initialize the engine (init time), which included pre-grounding in the incremental mode, (ii) the average time per increment (tick time), and (iii) the total time of a single run (total time), which resulted from init time and tick time for all atoms and time points. It is important to note that adding or removing rules may be involved in a tick increment (tick time). We evaluated both reasoners. For Ticker, we evaluated the simulated datasets once with JTMS (Ticker), the original version of Ticker, and once with ATMS (LEA). The evaluations were conducted on a laptop equipped with an Intel i7 CPU and 16 GB RAM, using JVM version 1.8.0 112. Further details and the code written in Scala, is available on GitHub at LEA.

Results. The results shown in Tables II, III, IV, V and VI reveal insightful information about the performance of different implementations and the influence of window size and time points on execution times in both scenarios. When comparing LEA, Ticker, and Laser implementations, the performance is almost equivalent in terms of execution time, with slightly lower scores for LEA (cf. RQ1). Indeed, by taking into consideration the tuple parameter and fixing the time point tp at 1000, we observe that LEA's total time slightly exceeded the one of Ticker and Laser in scenarios A (maximum total time 1.857 seconds) and B (maximum total time 0.623 seconds)(cf. RQ4). Additionally, when fixing the window size to n = 60, we observed the same results. This could be explained by the high degree of interdependence among the LARS rules, which generates more justifications in the ATMS graph. Consequently, the *PROPAGATE* function becomes costly. Although LEA consumes more execution time than Ticker and Laser, it is the only solution that provides powerful expression quality (cf. RQ2) by handling contradiction and performing calculations on double arguments using the proposed LARS extension and Algorithm 1 (see TableVI). Moreover, comparing both scenarios, the nature of the rules themselves influences incremental evaluation and grounding. Complex rules with intricate dependencies and nested structures (Scenario B) can result in longer evaluation time and more extensive grounding operations (cf. RQ3).

TABLE II
RESULTS FOR SCENARIO A. VARIABLE WINDOW SIZE N. RESULTS FOR FIXED TIMEPOINTS 1000 AND TUPLE INSTANCE IN SECONDS.

impl	winsize n	total time	init time	tick time
Ticker	20	0.468	0.02	0.008
	40	0.494	0.01	0.009
	80	0.51	0.018	0.009
	120	0.744	0.017	0.014
	160	0.832	0.026	0.015
	200	0.744	0.018	0.014
Laser	20	0.568	0.03	0.018
	40	0.594	0.02	0.019
	80	0.61	0.028	0.019
	120	0.844	0.027	0.024
	160	0.932	0.036	0.025
	200	0.844	0.028	0.024
LEA	20	0.807	0.032	0.072
	40	0.985	0.015	0.022
	80	0.565	0.009	0.027
	120	0.334	0.01	0.035
	160	0.397	0.012	0.041
	200	1.857	0.011	0.02

TABLE III
RESULTS FOR SCENARIO A.VARIABLE TIME POINTS TP. RESULTS FOR FIXED WINDOW SIZE N = 60 AND TUPLE INSTANCE IN SECONDS.

impl	tp	total time	init time	tick time
Ticker	100	0.096	0.015	0.008
	200	0.202	0.008	0.009
	300	0.226	0.015	0.007
	400	0.405	0.017	0.009
	500	0.462	0.021	0.008
	600	0.561	0.009	0.009
	700	0.775	0.013	0.01
	800	0.781	0.009	0.009
	900	1.086	0.026	0.011
	1000	1.272	0.018	0.012
Laser	100	0.115	0.018	0.01
	200	0.242	0.01	0.011
	300	0.271	0.018	0.008
	400	0.486	0.02	0.011
	500	0.554	0.025	0.01
	600	0.673	0.011	0.011
	700	0.93	0.016	0.012
	800	0.937	0.011	0.011
	900	1.303	0.031	0.013
LEA	100	0.247	0.013	0.015
	200	0.564	0.016	0.018
	300	1.004	0.014	0.02
	400	1.765	0.016	0.026
	500	2.088	0.01	0.022
	600	2.685	0.013	0.024
	700	3.51	0.011	0.024
	800	4.196	0.011	0.025
	900	5.269	0.02	0.028
	1000	6.068	0.014	0.029

VI. CONCLUSION

We proposed in this paper LEA, a new stream reasoning engine based on Ticker that overcomes its limitations. We have

TABLE IV
RESULTS FOR B WITH FIXED TIMEPOINTS TP = 1000 AND TUPLE INSTANCE IN SECONDS.

impl	winsize n	total time	init time	tick time
Ticker	20	0.359	0.25	0.01
	40	0.359	0.259	0.01
	80	0.37	0.262	0.01
	120	0.354	0.246	0.01
	160	0.367	0.251	0.011
	200	0.364	0.266	0.008
Laser	20	0.431	0.3	0.012
	40	0.431	0.311	0.012
	80	0.444	0.314	0.012
	120	0.425	0.295	0.012
	160	0.44	0.301	0.013
	200	0.437	0.319	0.01
LEA	20	0.541	0.198	0.025
	40	0.559	0.172	0.03
	80	0.611	0.196	0.032
	120	0.615	0.181	0.033
	160	0.623	0.196	0.031
	200	0.54	0.196	0.026

TABLE V
RESULTS FOR SCENARIO B WITH FIXED WINDOW SIZE N = 60 IN SECONDS

impl	tp	total time	init time	tick time
Ticker	100	0.355	0.236	0.011
	200	0.401	0.235	0.008
	300	0.413	0.208	0.006
	400	0.468	0.23	0.005
	500	0.51	0.216	0.005
	600	0.596	0.225	0.005
	700	0.703	0.243	0.006
	800	0.86	0.255	0.006
	900	0.93	0.227	0.007
	1000	1.09	0.211	0.008
Laser	100	0.426	0.283	0.013
	200	0.481	0.282	0.01
	300	0.496	0.25	0.007
	400	0.562	0.276	0.006
	500	0.612	0.259	0.006
	600	0.715	0.27	0.006
	700	0.844	0.292	0.007
	800	1.026	0.282	0.006
	900	1.116	0.272	0.008
	1000	1.308	0.253	0.01
LEA	100	1.255	0.478	0.068
	200	1.734	0.451	0.053
	300	2.113	0.416	0.045
	400	2.752	0.379	0.043
	500	3.762	0.4	0.048
	600	4.772	0.401	0.051
	700	5.816	0.371	0.055
	800	6.414	0.397	0.051
	900	8.995	0.385	0.064
	1000	9.794	0.366	0.063

extended the LARS formula to express rule contradiction within a time window. We have also increased the expressiveness of the rules by providing the ability to use double arguments and string comparison. In addition, facts are processed on-the-fly and do not need to be specified upstream for each rule. With this extension, we have fulfilled requirements R1, R2 and R3. Finally, to satisfy requirement R4, instead of using JTMS, we propose a new implementation of ATMS. The conducted evaluation highlights the strengths of LEA and emphasizes its potential for handling complex reasoning scenarios in real-time data streams. As future work, we plan to optimize the PROPAGATE function to enhance LEA's capabilities. We will also address the issue of constraints management in stream reasoning engines and in particular in ATMS-stored constraints.

TABLE VI
COMPARISON OF RESULT QUALITY WITH DIFFERENT IMPLEMENTATIONS

Impl	Scenario	$\perp$	Double Args	Operation on Args
Ticker	A	×	×	×
	B	×	×	×
Laser	A	×	×	×
	B	×	×	×
LEA	A	✓	✓	✓
	B	✓	✓	✓

REFERENCES

[1] E. D. Valle, S. Ceri, F. van Harmelen, and D. Fensel, "It's a streaming world! reasoning upon rapidly changing information," *IEEE Intell. Syst.*, vol. 24, no. 6, pp. 83–89, 2009. [Online]. Available: https://doi.org/10.1109/MIS.2009.125

[2] H. Beck, M. Dao-Tran, and T. Eiter, "Lars: A logic-based framework for analytic reasoning over streams," *Artificial Intelligence*, vol. 261, pp. 16–70, 2018.

[3] H. Beck, T. Eiter, and C. Folie, "Ticker: A system for incremental asp-based stream reasoning," *Theory and Practice of Logic Programming*, vol. 17, no. 5-6, pp. 744–763, 2017.

[4] H. R. Bazoobandi, H. Beck, and J. Urbani, "Expressive stream reasoning with laser," in *The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Proceedings, Part I*. Springer, 2017, pp. 87–103.

[5] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko, "Theory solving made easy with clingo 5," in *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[6] H. Beck, "Reviewing justification-based truth maintenance systems from a logic programming perspective."

[7] J. Doyle, "A truth maintenance system," *Artificial intelligence*, vol. 12, no. 3, pp. 231–272, 1979.

[8] M. I. Ali, F. Gao, and A. Mileo, "Citybench: A configurable benchmark to evaluate RSP engines using smart city datasets," in *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, Eds., vol. 9367. Springer, 2015, pp. 374–389. [Online]. Available: https://doi.org/10.1007/978-3-319-25010-6_25

[9] J. de Kleer, "A general labeling algorithm for assumption-based truth maintenance," in *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*, H. E. Shrobe, T. M. Mitchell, and R. G. Smith, Eds. AAAI Press / The MIT Press, 1988, pp. 188–192. [Online]. Available: http://www.aaai.org/Library/AAAI/1988/aaai88-034.php

[10] D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein, "Stream reasoning: A survey and outlook," *Data Science*, vol. 1, no. 1-2, pp. 59–83, 2017.

[11] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo," *Theory Pract. Log. Program.*, vol. 19, no. 1, pp. 27–82, 2019. [Online]. Available: https://doi.org/10.1017/S1471068418000054

[12] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub, "Stream reasoning with answer set programming: Preliminary report," in *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press, 2012. [Online]. Available: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4504