Towards a Logic-Based Framework for Analyzing Stream Reasoning*

Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{beck,dao,eiter,fink}@kr.tuwien.ac.at

Abstract. The rise of smart applications has drawn interest to logical reasoning over data streams. Recently, different query languages and stream processing/reasoning engines were proposed in different communities. However, due to a lack of theoretical foundations, the expressivity and semantics of these diverse approaches were given only informally. Towards clear specifications and means for analytic study, a formal framework is needed to define their semantics in precise terms. To this end, we present ongoing work towards such a framework and show how a core fragment of the prominent Continuous Query Language can be captured.

1 Introduction

The emergence of sensors, networks, and mobile devices has generated a trend towards *pushing* rather than *pulling* of data in information processing. In the setting of *stream processing* [1] studied by the database community, input tuples dynamically arrive at systems in form of possibly infinite streams. To deal with unboundedness of data, respective systems typically apply *window operators* to obtain snapshots of recent data. On these, the user runs *continuous queries* which are triggered either periodically or by events, e.g., by the arrival of new input. A well-known stream processing language is the Continuous Query Language (CQL) [2] which has an SQL-like syntax and a clear operational semantics.

Recently, the rise of *smart applications* such as smart cities, smart home, smart grid, etc., has raised interest in the topic of *stream reasoning* [3], i.e., logical reasoning on streaming data. Consider the following example from the public transport domain.

Example 1. To monitor a city's public transportation, the city traffic center has a background data set regarding the assignment of trams to lines of the form $line(ID, L)$, where ID is the tram identifier and L the identifier of the line. The planned travelling time (duration Z) between stops X and Y with line L is stored in a database by a row $plan(L, X, Y, Z)$. Old trams which are not convenient for travelling with wheel chairs or baby strollers are classified with facts of the form $old(ID)$. Moreover, sensor data of the form $tram(ID, X)$ reports the appearance of tram ID at stop X.

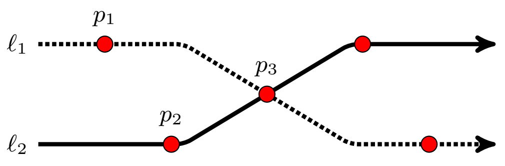
On top of these background data and streaming tuples, one may provide smart services that report the traffic status and suggest updates for travel routes in real time. For

* This research has been supported by the Austrian Science Fund (FWF) project P26471.

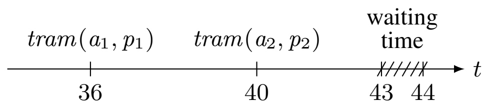
PLAN			
L	X	Y	Z
ℓ_1	p_1	p_3	8
ℓ_2	p_2	p_3	3
...			

LINE	
ID	L
a_1	ℓ_1
a_2	ℓ_2
...	

OLD
a_1
...



(a) Public transportation map



(b) Time line

Fig. 1: Traffic scenario

instance, it is of interest whether the service is reliable, i.e., whether at a given stop a tram appeared within the last 10 minutes. In addition, a user usually wants to know the expected arrival time of the next tram. Moreover, consider the case that the user travelling with a baby stroller has different options to change trams to reach a goal. She prefers to have short waiting time, and to switch to trams that are convenient for the baby stroller, which are not the old ones. Thus, we are interested in services that report about: (s_1) the *reliability* of a tram line at a stop, (s_2) the *expectation* about tram arrival times, and (s_3) the quality of the *connection* between two lines at a given stop.

Fig. 1a shows a simple map with two tram lines ℓ_1 and ℓ_2. The background data comprises three tables. PLAN states the planned travelling time between the stops, LINE shows the assignments of trams to lines, and OLD identifies old trams. Consider the scenario of Fig. 1b which depicts tram arrival. Let the time granularity be minutes, the reading of predicate $tram(a_1, p_1)$ at $t = 36$ means that tram a_1 arrived at stop p_1 at minute 36. Similarly, tram a_2 arrived at stop p_2 at minute 40. Based on this input stream and the background data, services (s_1)–(s_3) provide the following reports:

(s_1) At minute 40, line ℓ_1 is reliable at stop p_1 but not at p_3, because in the last 10 minutes, a_1 appeared at p_1 but no other tram of line ℓ_1 appeared at p_3.

(s_2) By the first row of PLAN, tram a_1 is expected to arrive at p_3 at minute 44. Then, by the second row, a_2 should arrive at p_3 one minute earlier, i.e., at minute 43.

(s_3) The option to switch from line ℓ_2 to ℓ_1 at p_3 satisfies the short waiting time requirement. However, since tram a_1 is old, it is not a good connection. ■

Different communities have contributed to different aspects of this topic. (i) The Semantic Web community extends SPARQL to allow querying on streams of RDF triples. Engines such as CQELS [4] and C-SPARQL [5] also follow the snapshot semantics approach of CQL. (ii) In Knowledge Representation and Reasoning (KRR), first attempts towards expressive stream reasoning have been carried out by considering continuous data in Answer Set Programming (ASP) [6, 7] or extending Datalog to sequential logic programs [8]. However, the state of the art in either field has several shortcomings.

Approaches in (i) face difficulties with extensions of the formalism to incorporate the Closed World Assumption, non-monotonicity, or non-determinism. Such features are important to deal with missing or incomplete data, which can, e.g., temporarily happen due to unstable network connections or hardware failure. In this case, engines like C-SPARQL and CQELS remain idle, while some output based on default reasoning might be useful. Moreover, e.g., in the use case of dynamic planning on live data, multiple plans shall be generated based on previous choices and the availability of new data. This is not possible with current deterministic approaches.

On the other hand, advanced reasoning has extensively been investigated in (ii) but traditionally only on static data. First attempts towards stream reasoning reveal many problems to solve. The plain approach of [6] periodically calls the dlvhex solver [9] without incremental reasoning and thus is not capable of handling heavy data load. StreamLog [8] is an extension of Datalog towards stream reasoning. It always computes a single model. OSMS [10] considers streams of ontologies. Both StreamLog and OSMS do not consider window mechanisms. ETALIS [16] provides a rule-based language for pattern matching over event streams with declarative monotonic semantics. Windows are also not first-class citizens in ETALIS. Time-decaying logic programs [7] are an attempt to implement time-based windows in reactive ASP [11] but its relation to other stream processing/reasoning approaches has not yet been explored.

Moreover, as observed in [12], conceptually identical queries may produce different results in different engines. While such deviations may occur due to differences (i.e., flaws) in implementations of a common semantics, they might also arise from (correct implementations of) different semantics. For a user it is important to know the exact capabilities and the semantic behavior of a given approach. However, there is a lack of theoretical underpinning or a formal framework for stream reasoning that allows to capture different (intended) semantics in precise terms. Investigations of specific languages, as well as comparisons between different approaches, are confined to experimental analysis [13], or informal examination on specific examples. A systematic investigation, however, requires a formalism to rigorously describe the expressivity and the properties of a language.

Contributions. In [14], we presented a first step towards a *formal framework for stream reasoning* that provides a common ground to express concepts from different stream processing/reasoning formalisms and engines. This allows for systematic analysis and comparison between existing stream processing/reasoning semantics. The idea of capturing related systems was informally discussed. This paper establishes the next step towards these goals by (i) generalizing further the notion of window functions, (ii) extending the formalization by rules, and (iii) formally capturing a core language of CQL.

We are interested in a stream reasoning semantics which is idealized in the sense that no information needs to be dropped but can be abstracted away. In particular, we allow to distinguish notions of truth of a formula at (i) specific time points, (ii) some time point in a window, or (iii) all time points in a window. Second, we idealize with respect to implementations and do not consider processing time, delays or outages in the semantics itself.

2 Streams

In this section, we introduce streams and generalize the window functions of [14].

2.1 Streaming Data

We build upon mutually disjoint sets of *predicates* $\mathcal{P}$ and *constants* $\mathcal{C}$. The set $\mathcal{A}$ of *atoms* is defined as $\{p(c_1, \ldots, c_n) \mid p \in \mathcal{P},\, c_1, \ldots, c_n \in \mathcal{C}\}$. If $i, j \in \mathbb{N}$, we call the set $[i, j] = \{k \in \mathbb{N} \mid i \leq k \leq j\}$ an *interval*. We divide the set $\mathcal{P}$ into two disjoint subsets, namely the *extensional predicates* $\mathcal{P}_\mathcal{E}$ and the *intensional predicates* $\mathcal{P}_\mathcal{I}$. The former is used for input streams and background data, while the latter serves for intermediate and output streams. Additionally, we assume basic arithmetic operations $(+, -, \times, \div)$ and comparisons $(=, \neq, <, >, \leq, \geq)$ to be predefined by designated predicates $\mathcal{B} \subseteq \mathcal{P}_\mathcal{E}$. For convenience, these predicates may be written in infix notation.

We now present the central notion of streams.

Definition 1 (Stream). *Let T be an interval and $\upsilon \colon \mathbb{N} \to 2^\mathcal{A}$ an* evaluation function *such that $\upsilon(t) = \emptyset$ for all $t \in \mathbb{N} \setminus T$. Then, the pair $S = (T, \upsilon)$ is called a* stream*, T is called the* timeline *of S and the elements of T are called* time points.

Let $S = (T, \upsilon)$ and $S' = (T', \upsilon')$ be two streams. We say S' is a *substream* or *window* of S', denoted $S' \subseteq S$, if $T \subseteq T'$ and $\upsilon'(t') \subseteq \upsilon(t')$ for all $t' \in T'$. Moreover, S' is a *proper substream* of S, denoted $S' \subset S$, if $S' \subseteq S$ and $S' \neq S$. The *size* $\#S$ of S is defined by $\Sigma_{t \in T} |\upsilon(t)|$. The *restriction* $S|_{T'}$ *of S to* $T' \subseteq T$, is the stream $(T', \upsilon|_{T'})$, where $\upsilon|_{T'}$ restricts the domain of υ to T', i.e., $\upsilon'(t) = \upsilon(t)$ for all $t \in T'$, else $\upsilon'(t) = \emptyset$. A stream is called *data stream* iff it contains only extensional predicates.

Example 2. The input for Example 1 can be modelled as a data stream $D = (T, \upsilon)$ where $T = [0, 50]$, $\upsilon(36) = \{\mathit{tram}(a_1, p_1)\}$, $\upsilon(40) = \{\mathit{tram}(a_2, p_2)\}$, and $\upsilon(t) = \emptyset$ for all $t \in T \setminus \{36, 40\}$. The granularity of T is minutes. The evaluation function υ can be equally represented as $\{36 \mapsto \{\mathit{tram}(a_1, p_1)\}, 40 \mapsto \{\mathit{tram}(a_2, p_2)\}\}$. ■

2.2 Windows

An essential aspect of stream reasoning is to restrict the considered data to so-called *windows*, i.e., recent substreams, in order to limit the amount of data and forget outdated information. Traditionally [2], these windows take a fixed size ranging backwards from query time. This size is determined in different ways. A *time-based* (or *sliding*) window contains all tuples of a fixed amount of time, e.g., the last five seconds. A *tuple-based* (or *count-based*) window, on the other hand, takes a fixed number of the most recent tuples, regardless of when they arrived. A more generic variant of the latter is the *partition-based* window, where a tuple-based window is applied on substreams obtained by the input stream. This way, the recent values for a specified list of attributes can be selected.

In [14] generalized notions of windows are presented. Based on a *reference time point* t, window functions can also collect tuples that arrive after t. A time-based (resp. tuple-based) window is then associated with fixed parameters ℓ and u that select the previous ℓ and the next u time units (resp. number of tuples) relative to t. For partition-based windows, another parameter is needed to specify how the input stream is split into substreams.

In order not to fix these parameters, but to be able to control the valid range of windows within rules, we take a more generic approach in the present work. In addition to a stream and a time point, window functions take a vector of further *window parameters* $\boldsymbol{x}$, which may contain constants (e.g. integers), time variables and functions.

Definition 2 (Window function). *A* window function w_ι *of* type ι *takes as input a stream* $S = (T, v)$*, a time point* $t \in T$*, called the* reference time point*, and a vector of* window parameters $\boldsymbol{x}$ *for type* ι *and returns a substream* S' *of* S.

Currently, the most common types of windows in practice are time-, tuple-, and partition-based windows. We associate them with three respective window functions w_τ, $w_\#$, and w_{p}. Their difference in the window parameters $\boldsymbol{x}$ and window building can be intuitively summarized as follows:

- *Time-based*: $\boldsymbol{x} = (\ell, u, d)$, where $\ell, u \in \mathbb{N} \cup \{\infty\}$ and $d \in \mathbb{N}$. The window function $w_\tau(S, t, \boldsymbol{x})$ returns the substream of S that contains all tuples of the last ℓ time units and the next u time units relative to a *pivot* time point t' derived from t and the step size d. We use $\ell = \infty$ (resp. $u = \infty$) to take all previous (resp. later) tuples.
- *Tuple-based*: $\boldsymbol{x} = (\ell, u)$, where $\ell, u \in \mathbb{N}$. The function $w_\#(S, t, \boldsymbol{x})$ selects a substream of S with the shortest interval $[t_\ell, t_u] \subseteq T$ as timeline, where $t_\ell \leq t \leq t_u$, such that ℓ tuples are in $[t_\ell, t]$ and u tuples are in $[t+1, t_u]$. Exactly ℓ, resp. u tuples are returned. In case of multiple options due to multiple tuples at time points t_ℓ, resp. t_u, only tuples from there are removed at random.
- *Partition-based*: $\boldsymbol{x} = (\mathrm{idx}, n)$ where idx and n are two total functions

$$\mathrm{idx}\colon \mathcal{A} \to I \subset \mathbb{N} \text{ and } n\colon I \to \mathbb{N} \times \mathbb{N}.$$

  Here, I is a non-empty, finite (index) set of integers. The application $w_{\mathrm{p}}(S, t, \boldsymbol{x})$ first splits the input stream $S = (T, v)$ into $|I|$ substreams $S_i = (T, v_i)$ by taking $v_i(t) = \{a \in v(t) \mid \mathrm{idx}(a) = i\}$. Then, a tuple-based window $w_\#$ is applied on each respective S_i with parameters taken from $n(i) = (\ell_i, u_i)$. The output streams after $w_\#$ are then merged to produce the resulting window.

Here, we gave a slight generalization of window functions as presented (more formally) in [14], using the general parameter vector $\boldsymbol{x}$. This will be more useful when we discuss dynamic window applications below.

Due to space reason, we present only the adaptation of time-based windows formally. The same idea can be applied to other types of windows straightforwardly.

Definition 3 (Time-based window). *Let* $S = (T, v)$ *be a stream,* $T = [t_{min}, t_{max}]$ *and* $t \in T$*. Moreover, let* $\boldsymbol{x} = (\ell, u, d)$ *where* $\ell, u \in \mathbb{N} \cup \{\infty\}$ *and* $d \in \mathbb{N}$ *s.t.* $d \leq \ell + u$. *The* time-based window with range (ℓ, u) and step size d of S at time t *is defined by*

$$w_\tau(S, t, \boldsymbol{x}) = (T', v|_{T'}),$$

where $T'{=}[t_\ell, t_u]$*,* $t_\ell = \max\{t_{min}, t'{-}\ell\}$ *with* $t' = \lfloor \frac{t}{d} \rfloor \cdot d$*, and* $t_u = \min\{t'{+}u, t_{max}\}$.

Example 3 (cont'd). To express the monitoring over the stream D of Example 2, one can use a time-based window function $w_\tau(D, t, (10, 0, 1))$ ranging over the past 10 minutes and step size of 1 minute. The result of applying this function at $t = 40$ is

$$w_\tau(D, 40, (10, 0, 1)) = ([30, 40], \{36 \mapsto \{\mathit{tram}(a_1, p_1)\}, 40 \mapsto \{\mathit{tram}(a_2, p_2)\}\}).$$

To get the last tram appearance, we can use a tuple-based window $w_\#(D, t, (1, 0))$. For example: $w_\#(D, 40, (1, 0)) = ([40, 40], \{40 \mapsto \{\mathit{tram}(a_2, p_2)\}\})$. ■

3 Logical Framework

We now introduce a logical framework for reasoning over streams. In addition to [14], the present work allows for rule-based programs. For their evaluation, we will make use of *interpretation streams* which augment data streams by intensional predicates. The semantics of programs is then defined by an entailment relation for formulas. The underlying logical language includes various operators to deal with reference to time.

3.1 Window operators

In what follows, we present a semantics for reasoning over streams, where we make use of window operators $\boxplus^{\boldsymbol{x}}_{\iota,ch}$ that work on *two* streams. This allows us to distinguish between a given stream S_1 and the currently considered window S_2. Therefore, before the application $w_\iota(S,t,\boldsymbol{x})$ of a window function, a *stream choice* function ch determines a stream S based on two streams S_1 and S_2. We use two simple stream choices $ch_i(S_1,S_2)=S_i$, where $i \in \{1,2\}$.

Definition 4 (Window operator). *Let w_ι be a window function of type ι, let ch be a stream choice function and let $\boldsymbol{x}$ be a vector of window parameters for type ι. Then, $\boxplus^{\boldsymbol{x}}_{\iota,ch}$ denotes a* window operator.

For more convenience, we omit writing ch_2. That is, by the window operator of form $\boxplus^{\boldsymbol{x}}_{\iota}$, we mean $\boxplus^{\boldsymbol{x}}_{\iota,ch_2}$. Moreover, we use special syntax for typical parameters $\boldsymbol{x}$ for tuple-based windows, and for time-based windows with step size $d=1$. For both types, we write only ℓ for $\boldsymbol{x}$, if $u=0$, and $+u$, if $\ell=0$. Consider the following examples:

$$\boxplus^{10}_{\tau} = \boxplus^{10,0,1}_{\tau,ch_2} \qquad \boxplus^{+5}_{\tau} = \boxplus^{0,5,1}_{\tau,ch_2} \qquad \boxplus^{1}_{\#} = \boxplus^{1,0}_{\#,ch_2}$$

Here, all operators extract tuples from the second stream. The symbol $\boxplus^{10}_{\tau}$ abbreviates the time-based window operator that takes all tuples of the last 10 time points and $\boxplus^{+5}_{\tau}$ takes all tuples of the next 5 time points. On the other hand, $\boxplus^{1}_{\#}$ takes the latest tuple which arrived until the reference time point. The relationship between window functions (Def. 2) and window operators (Def. 4) will be made precise in Def. 7 below.

3.2 Formulas

Syntax. In addition to the window operators $\boxplus^{\boldsymbol{x}}_{\iota,ch}$, we make use of further means to refer to or abstract from time. Similarly as in modal logic, we will use operators $\Box$ and $\Diamond$ to test whether a tuple (atom) or a formula holds all the time, respectively sometime in a window. Moreover, we use an *exact* operator @ to refer to specific time points.

Definition 5 (Formulas). *The set $\mathcal{F}$ of* formulas *is defined by the grammar*

$$\alpha ::= a \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \alpha \rightarrow \alpha \mid \Diamond\alpha \mid \Box\alpha \mid @_t\alpha \mid \boxplus^{\boldsymbol{x}}_{\iota,ch}\alpha \qquad (1)$$

where a is any atom in $\mathcal{A}$, $t \in \mathbb{N}$ and $\boxplus^{\boldsymbol{x}}_{\iota,ch}$ is a window operator.

Intuitively, given a stream $S^\star$ and a considered window S (which initially is $S^\star$), each formula α will be evaluated based on a reference time point t within S. An application of a window operator $\boxplus^{\boldsymbol{x}}_{\iota,ch}$ creates a new window S', which depends on $S^\star$ and S as specified by the stream choice. Within the current window S, $\Diamond\alpha$ (resp. $\Box\alpha$) holds, if α holds at some time point (resp. at all time points) in S. Relative to t, the formula α holds if α is true exactly at t, and $@_{t'}\alpha$ holds if t' is in the timeline of S and α is true exactly at time point t'. That is, the operator @ allows to jump to a specific time point within the window.

Semantics. In addition to streams, we consider background knowledge in form of a static data set, i.e., a set of atoms which does not change over time. From a semantic perspective, the difference to streams is that static data is always available, regardless of window applications.

Definition 6 (Structure). *Let $S = (T, \upsilon)$ be a stream, W be a set of window functions and $B \subseteq \mathcal{A}$ a set of facts. Then, we call $M = \langle T, \upsilon, W, B\rangle$ a* structure*, S the* interpretation stream *and B the* data set *or* background data *of M.*

We now define when a formula holds in a structure.

Definition 7 (Entailment). *Let $M = \langle T^\star, \upsilon^\star, W, B\rangle$ be a structure, $S^\star = (T^\star, \upsilon^\star)$ and let $S = (T, \upsilon)$ be a substream of $S^\star$. Moreover, let $t \in T$. The* entailment *relation $\Vdash$ between (M, S, t) and formulas is defined as follows. Let $a \in \mathcal{A}$ be an atom, and let $\alpha, \beta \in \mathcal{F}$ be formulas. Then,*

$$\begin{array}{lll}
M,S,t \Vdash a & \textit{iff} & a \in \upsilon(t) \textit{ or } a \in B,\\
M,S,t \Vdash \neg\alpha & \textit{iff} & M,S,t \nVdash \alpha,\\
M,S,t \Vdash \alpha\wedge\beta & \textit{iff} & M,S,t \Vdash \alpha \textit{ and } M,S,t \Vdash \beta,\\
M,S,t \Vdash \alpha\vee\beta & \textit{iff} & M,S,t \Vdash \alpha \textit{ or } M,S,t \Vdash \beta,\\
M,S,t \Vdash \alpha\rightarrow\beta & \textit{iff} & M,S,t \nVdash \alpha \textit{ or } M,S,t \Vdash \beta,\\
M,S,t \Vdash \Diamond\alpha & \textit{iff} & M,S,t' \Vdash \alpha \textit{ for some } t' \in T,\\
M,S,t \Vdash \Box\alpha & \textit{iff} & M,S,t' \Vdash \alpha \textit{ for all } t' \in T,\\
M,S,t \Vdash @_{t'}\alpha & \textit{iff} & M,S,t' \Vdash \alpha \textit{ and } t' \in T,\\
M,S,t \Vdash \boxplus^{\boldsymbol{x}}_{\iota,ch}\alpha & \textit{iff} & M,S',t \Vdash \alpha \textit{ where } S' = w_\iota(ch(S^\star,S),t,\boldsymbol{x}).
\end{array}$$

If $M, S, t \Vdash \alpha$ holds, we say that (M, S, t) *entails* α.

Example 4 (cont'd). Let $D = (T, \upsilon)$ be the data stream of Ex. 3 and $S^\star = (T^\star, \upsilon^\star) \supseteq D$ be a stream such that $T^\star = T$ and

$$\upsilon^\star = \left\{ \begin{array}{ll} 36 \mapsto \{\mathit{tram}(a_1, p_1)\}, & 40 \mapsto \{\mathit{tram}(a_2, p_2), \mathit{rel}(a_1, p_1)\}, \\ 43 \mapsto \{\mathit{exp}(a_2, p_3)\}, & 44 \mapsto \{\mathit{exp}(a_1, p_3)\} \end{array} \right\}.$$

Let $M = \langle T^\star, \upsilon^\star, W, B\rangle$ where $W = \{w_\tau, w_\#, w_\mathrm{p}\}$ and let B be the background data from Example 1. We see that $M, S^\star, 43 \Vdash \boxplus^{+5}_\tau \Diamond \mathit{tram}(a_2, p_3)$ holds. Indeed, the window operator $\boxplus^{+5}_\tau$ selects the substream $S' = (T', \upsilon')$ where $T' = [43, 48]$, and

$$\upsilon' = \big\{\, 43 \mapsto \{\mathit{exp}(a_2, p_3)\},\ 44 \mapsto \{\mathit{exp}(a_1, p_3)\} \,\big\}.$$

Next, because $\mathit{exp}(a_1, p_3) \in \upsilon'(43)$ and $43 \in T'$, it holds that $M, S', 43 \Vdash \Diamond\mathit{exp}(a_2, p_3)$. Similarly, we have $M, S^\star, 40 \Vdash w^{10}_\tau \Diamond \mathit{tram}(a_1, p_1)$ holds. ■

3.3 Programs

Based on the entailment relation, we are now going to define a rule-based semantics.

Definition 8 (Rule, Program). *A* program *is a set of* rules*, i.e., expressions of the form*

$$\alpha \leftarrow \beta_1, \ldots, \beta_j, \mathrm{not}\, \beta_{j+1}, \ldots, \mathrm{not}\, \beta_n \ , \tag{2}$$

where $\alpha, \beta_1, \ldots, \beta_n \in \mathcal{F}$ are formulas and α contains only intensional predicates.

Let D be a data stream based on which we want to evaluate a program P. Moreover, let $I{=}(T, \upsilon)$ be a stream such that $D \subseteq I$. If for all time points in T, all predicates that occur in I but not in D are intensional, then we call I an *interpretation stream for D* and a structure $M = \langle T, \upsilon, W, B\rangle$ an *interpretation (for D)*. We say that M *satisfies P at time t*, denoted by $M, t \models P$, if (M, I, t) entails each rule $r \in P$ viewed as classical implication $\beta(r) \rightarrow \alpha$, where $\beta(r) = \beta_1 \wedge \ldots \wedge \beta_j \wedge \neg\beta_{j+1} \wedge \ldots \wedge \neg\beta_n$. In this case, we say M is a *model (of P for D at time t)*. We call M a *minimal model*, if no model $M'{=}\langle T', \upsilon', W, B\rangle$ of P for D at time t exists such that $(T', \upsilon') \subset (T, \upsilon)$. The *reduct* of a program P w.r.t. M at time t is defined by $P^{M,t} = \{r \in P \mid M, t \models \beta(r)\}$.

Definition 9 (Answer). *Let P be a program and D be a data stream. The interpretation $M = \langle T, \upsilon, W, B\rangle$ for D is called an* answer *of P for D at time t if M is a minimal model of the reduct $P^{M,t}$.*

Note that the notion of an answer is relative to fixed functions W and static background data B.

Towards more conciseness, we consider schematic programs with variables of two sorts, namely constant variables and time variables. The semantics of these *non-ground programs* is given by the answers of according groundings (obtained by replacing variables with constants from $\mathcal{C}$, respectively time points from T, in all possible ways).

Example 5. The requests (s_1)–(s_3) from Example 1 can be formulated by rules (3)–(5).

$$\mathit{rel}(L, X) \leftarrow \mathit{line}(ID, L), \boxplus_{\tau}^{10} \Diamond \mathit{tram}(ID, X). \tag{3}$$

$$\begin{aligned} @_T\, \mathit{exp}(ID, Y) \leftarrow\ & \boxplus_{\mathrm{p}}^{\mathrm{idx},n}\ @_{T_1} \mathit{tram}(ID, X), \\ & \mathit{line}(ID, L), \mathit{plan}(L, X, Y, Z), T = T_1 + Z. \end{aligned} \tag{4}$$

$$\mathit{gc}(ID_1, ID_2, X) \leftarrow @_T\, \mathit{exp}(ID_1, X), @_T \boxplus_{\tau}^{+5} \Diamond \mathit{exp}(ID_2, X), \mathrm{not}\ \mathit{old}(ID_2). \tag{5}$$

Rule (3) uses the time-based window operator that checks whether a tram of line L appeared within the last 10 minutes to conclude whether the line is reliable. Rule (4) calculates when a tram is expected at later stops. To get the last appearances of every individual tram instance from the input stream, we use the partition-based window operator $\boxplus_{\mathrm{p}}^{\mathrm{idx},n}$, where the two functions idx and n are defined by

$$\begin{aligned} &\mathrm{idx}(\mathit{tram}(a_i, X)) = i \quad \text{and} \quad \mathrm{idx}(a, X) = 0 \text{ for all } a \in \mathcal{A} \setminus \{\mathit{tram}(a_i, X)\}; \\ &n(i) = (1, 0) \text{ for } i > 0 \quad \text{and} \quad n(0) = (0, 0). \end{aligned}$$

Finally, rule (5) exploits the result from rule (4) to identify good connections where the targeted tram is not old and the expected waiting time is at most 5 minutes. This rule uses a time-based window that looks forward 5 mins from the time when $\mathit{exp}(ID_1, X)$ is concluded and the operator $\Diamond$ to check the existence of an expected tram instance. One can check that the structure M in Example 4 is an answer of P. ■

4 Capturing CQL

The Continuous Query Language (CQL) [2] is an SQL based language for maintaining continuous queries over streaming data. Such queries are suitable for real-time and reactive programs. To handle input streams, CQL extends SQL with three sets of operators:

(i) *Stream-to-relation* (S2R) operators apply window functions to the input stream to create a mapping from execution times to bags of valid tuples (w.r.t. the window) without timestamps. This mapping is called a relation.
(ii) *Relation-to-relation* (R2R) operators allow for modification of relations similarly as in relational algebra, respectively SQL.
(iii) *Relation-to-stream* (R2S) operators convert relations to streams by directly associating the timestamp of the execution with each tuple (RStream). The other operators IStream/DStream, which report inserted/deleted tuples, are derived from RStream.

Example 6. Let `PLAN(ID,X,Y,Z)`, `LINE(ID,L)`, and `OLD(ID)` be three relations that correspond to the background data in Example 1. Let `TRAM(ID,X,T1)` be an input stream of tram appearances detected at respective tram stops. Compared to the input stream S in Example 1, the schema of `TRAM` carries an additional field for explicit timestamps of the input tuples. The following CQL queries serves the requests (s_1) and (s_2) in Example 1:

```
q1(rel) = SELECT L, X FROM TRAM [RANGE 5], LINE
          WHERE  TRAM.ID = LINE.ID

q2(exp) = SELECT TRAM.ID, PLAN.Y, T2
          FROM   TRAM [PARTITION BY ID ROWS 1], LINE, PLAN
          WHERE  TRAM.ID = LINE.ID AND LINE.L = PLAN.L AND
                 TRAM.X = PLAN.X AND T2 = TRAM.T1 + PLAN.Z  ■
```

We now present an approach for capturing CQL using the presented logic-based framework. In this paper, we consider a core fragment of CQL without nested queries:
− The `FROM` clause is a sequence of only input streams with windows, or background data table names. In that sense, only the keyword `AND` is used for connecting the elements of the `FROM` clauses. Furthermore, renaming the above input sources using the keyword `AS` is also left out.
− The `WHERE` clause may contain only plain arithmetic operators and comparisons. Nested expressions can always be rewritten into multiple such plain operators. For instance, `X<Y+Z` can be expressed as `W=Y+Z AND X<W`. Generally speaking, the conditions in the `WHERE` clause are of the form

(a) `P1.X1` *cmp* `P2.X2`, or
(b) `P1.X1` = `P2.X2` *op* `P3.X3`,

where `P1`, `P2`, `P3` are streams or background data table names, `X1`, `X2`, `X3` are attribute names, $cmp \in \{=, \neq, <, >, \leq, \geq\}$ and $op \in \{+, -, \times, \div\}$. Without loss of

Element f in `FROM` clause	$\mathit{\Delta}_{\mathsf{FROM}}(f)$
`S`	$S(\boldsymbol{X})$
`S [RANGE L]`	$\boxplus_{\tau}^{L}\Diamond S(\boldsymbol{X})$
`S [RANGE L SLIDE D]`	$\boxplus_{\tau}^{L,0,D}\Diamond S(\boldsymbol{X})$
`S [RANGE UNBOUNDED]`	$\boxplus_{\tau}^{\infty}\Diamond S(\boldsymbol{X})$
`S [NOW]`	$S(\boldsymbol{X})$
`S [ROWS N]`	$\boxplus_{\#}^{N}\Diamond S(\boldsymbol{X})$
`S [PARTITION BY X1,X2,...,Xk ROWS N]`	$\boxplus_{\mathrm{p}}^{\mathrm{idx},n}\Diamond S(\boldsymbol{X})$

Table 1: Translation for `FROM` clauses

generality, we assume for case (a) that only $=$ is used for cmp when `X1` and `X2` are identical, and for case (b) that all attribute names are different. This assumption is to avoid going into technical details about renaming of attribute names to deal with special cases such as comparison of the form `P1.X` $<$ `P2.X`, or expression of the form `P1.X` $=$ `P2.X` $+$ `P3.X`.
$-$ Post-processing functionalities such as `GROUP BY`, `ORDER BY` are omitted.

To capture the considered CQL core fragment, we introduce three translation functions to cover the three types of operators (i)-(iii) above, with the following intuition:

(i) For S2R operators, $\mathit{\Delta}_{\mathsf{FROM}}$ takes input streams and background data table names appearing in the `FROM` clause. For streams, $\mathit{\Delta}_{\mathsf{FROM}}$ translates them to rule bodies where window operators are applied on the input streams. For background data, $\mathit{\Delta}_{\mathsf{FROM}}$ simply generates a corresponding predicate for the rule body.
(ii) For R2R operators, $\mathit{\Delta}_{\mathsf{WHERE}}$ takes the arithmetic operators and comparisons in the `WHERE` clause to create corresponding built-in or comparison predicates.
(iii) For R2S operators, $\mathit{\Delta}_{\mathsf{SELECT}}$ creates a rule head with a single atom whose predicate name is the query name. Its arguments are from the attributes specified in the `SELECT` clause.

More formally, assume that S is the name of an input stream or a background data table whose schema is given by a vector $\boldsymbol{X}$ of parameters. Let $F_q = f_1, \dots, f_k$ be the list of elements of the `FROM` clause of a CQL query q. Then, we obtain the translation of F_q by $\mathit{\Delta}_{\mathsf{FROM}}(q) = \{\mathit{\Delta}_{\mathsf{FROM}}(f_i) \mid f_i \in F_q\}$ as in Table 1. (We assume that idx mimics the partitioning based on attributes $X_1, \dots, X_k$ of $\boldsymbol{X}$ to a finite index set $I \subset \mathbb{N}$ and $n(i) = (N, 0)$ for all $i \in I$.)

Example 7 (cont'd). We have the following `FROM` clause translations for Example 6:

$$\begin{aligned}
\mathit{\Delta}_{\mathsf{FROM}}(q_1) &= \{\boxplus_{\tau}^{5}\Diamond \mathit{tram}(ID, X, T_1), \mathit{line}(ID, L)\}, \\
\mathit{\Delta}_{\mathsf{FROM}}(q_2) &= \{\boxplus_{\mathrm{p}}^{\mathrm{idx},n}\Diamond \mathit{tram}(ID, X, T_1), \mathit{line}(ID, L), \mathit{plan}(ID, X, Y, Z)\}. \quad \blacksquare
\end{aligned}$$

For the `WHERE` clause of a query q, let $WH_q = wh_1, \dots, wh_\ell$ be the list of conditions of the form (a) or (b) in the `WHERE` clause of a query q. The translation $\mathit{\Delta}_{\mathsf{WHERE}}$ applied to wh_i, $i \leq 1 \leq \ell$ is as follows:

- $\Delta_{\mathsf{WHERE}}(\texttt{P1.X} = \texttt{P2.X}) = \emptyset$.
- $\Delta_{\mathsf{WHERE}}(\texttt{P1.X1}\ \mathit{cmp}\ \texttt{P2.X2}) = \mathit{cmp}(X_1, X_2)$, wherethe attribute names X1 and X2 are not identical.
- $\Delta_{\mathsf{WHERE}}(\texttt{P1.X1} = \texttt{P2.X2}\ \mathit{op}\ \texttt{P3.X3}) = \mathit{op}(X_2, X_3, X_1)$.

The translation of WH_q is $\Delta_{\mathsf{WHERE}}(q) = \{\Delta_{\mathsf{WHERE}}(wh_j) \mid wh_j \in \mathit{WH}_q\}$.

Example 8 (cont'd). Consider the WHERE clauses of Example 6. Its translations are given by $\Delta_{\mathsf{WHERE}}(q_1) = \emptyset$ and $\Delta_{\mathsf{WHERE}}(q_2) = \{T_2 = T_1 + Z\}$. ■

Finally, the translation Δ_{SELECT} of a query named q is $\Delta_{\mathsf{SELECT}}(q) = q(\boldsymbol{Y})$, where $\boldsymbol{Y}$ is the list of attribute names appearing in the SELECT clause. For SELECT $\star$, the list of parameters $\boldsymbol{Y}$ is the collection of all parameters from $\Delta_{\mathsf{FROM}}(q) \cup \Delta_{\mathsf{WHERE}}(q)$. Putting all together, given a CQL query q, the function Δ translates it into a rule of the form

$$\Delta(q) = \Delta_{\mathsf{SELECT}}(q) \leftarrow \Delta_{\mathsf{FROM}}(q) \cup \Delta_{\mathsf{WHERE}}(q). \tag{6}$$

The translation of a set $Q = \{q_1, \ldots, q_m\}$ of CQL queries is $\Delta(Q) = \{\Delta(q) \mid q \in Q\}$.

Example 9 (cont'd). For the queries from Example 6, we have the SELECT translations $\Delta_{\mathsf{SELECT}}(q_1) = \mathit{rel}(L, X)$ and $\Delta_{\mathsf{SELECT}}(q_2) = \mathit{exp}(\mathit{ID}, Y, T_2)$. Thus, we get

$$\begin{aligned} \Delta(q_1) &= \mathit{rel}(L, X) \leftarrow \boxplus^{10}_{\tau} \Diamond \mathit{tram}(\mathit{ID}, X, T_1), \mathit{line}(\mathit{ID}, L). \\ \Delta(q_2) &= \mathit{exp}(\mathit{ID}, Y, T_2) \leftarrow \boxplus^{\mathrm{idx},n}_{\mathrm{p}} \Diamond \mathit{tram}(\mathit{ID}, X, T_1), \mathit{line}(\mathit{ID}, L), \\ &\qquad \mathit{plan}(\mathit{ID}, X, Y, Z), T_2 = T_1 + Z. \end{aligned}$$

We observe that these rules are not identical to rules (3) and (4) from Example 5 because CQL needs to explicitly carry the timestamps of input tuples to perform operators such as $@$, $\Diamond$ after the application of windows. ■

The following proposition states that this translation faithfully captures the considered core fragment of CQL.

Proposition 1. *Let $Q = \{q_1, \ldots, q_m\}$ be a set of CQL queries and $P = \Delta(Q)$. There is a 1-1 correspondence between the results of Q and the answers of P.*

Discussion. We now give hints on how to extend this translation to a larger fragment of CQL. The most important operation is nesting queries. Capturing them is expected to succeed as usual, i.e., by generalizing mappings of SQL statements to stratified programs [15] where subqueries provide lower strata for evaluating the query at a higher stratum. Moreover, the combination of window operators with the operators $\Box$ and $\Diamond$, preceded by the negation operator not, can already capture subqueries that check for the non-existence of a value reading within a range defined by the window. Orthogonal to this is aggregation in CQL queries. For this, we need to extend programs with aggregations, which helps not only for capturing aggregations, but also for capturing the post-processing statement GROUP BY. Another statement, ORDER BY, needs further auxiliary rules for encoding the order explicitly, because in the answers of programs, the evaluation function υ maps time points to sets of atoms which impose no order. Thus, while technically involved, generalizing our result to capture CQL without noteworthy restrictions seems feasible and is subject of our ongoing work.

5 Conclusion

Towards a logic-based framework for analyzing stream reasoning, we extended the work in [14] with a generalized notion of window functions and a rule-based language. We demonstrated the capability of this framework by capturing a core fragment of CQL.

Next steps include deeper study regarding capturing CQL fully as well as stream processing/reasoning systems, and a systematic comparison between these systems. In relation to complex event processing, considering time intervals is an interesting research issue. To improve practicality (as a tool for formal and experimental analysis) one might develop an operational characterization of the framework. In a longer perspective, along the same lines as [17], we aim at a formalism for stream reasoning in distributed settings across heterogeneous nodes that have potentially different logical capabilities.

References

1. Babu, S., Widom, J.: Continuous queries over data streams. SIGMOD Record **3**(30) (2001) 109–120
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2) (2006) 121–142
3. Valle, E.D., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems **24** (2009) 83–89
4. Phuoc, D.L., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: ISWC (1). (2011) 370–388
5. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-SPARQL: a continuous query language for rdf data streams. Int. J. Semantic Computing **4**(1) (2010) 3–25
6. Do, T.M., Loke, S.W., Liu, F.: Answer set programming for stream reasoning. In: AI. (2011) 104–109
7. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming. preliminary report. In: KR. (2012) 613–617
8. Zaniolo, C.: Logical foundations of continuous query languages for data streams. In: Datalog. (2012) 177–189
9. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: dlvhex: A prover for semantic-web reasoning under the answer-set semantics. In: Web Intelligence. (2006) 1073–1074
10. Ren, Y., Pan, J.Z.: Optimising ontology stream reasoning with truth maintenance system. In: CIKM. (2011) 831–836
11. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: ICLP. (2008) 190–205
12. Dindar, N., Tatbul, N., Miller, R.J., Haas, L.M., Botan, I.: Modeling the execution semantics of stream processing engines with secret. VLDB J. **22**(4) (2013) 421–446
13. Phuoc, D.L., Dao-Tran, M., Pham, M.D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: Facts and figures. In: ISWC - ET. (2012)
14. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: Towards Ideal Semantics for Analyzing Stream Reasoning. In: ReactKnow. (2014)
15. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)
16. Anicic, D., Rudolph, S., Fodor, P., Stojanovic., N.: Stream reasoning and complex event processing in ETALIS. Semantic Web Journal (2012)
17. Brewka, G.: Towards reactive multi-context systems. In: LPNMR. (2013) 1–10

A Appendix

A.1 Proof Sketch of Proposition 1

The idea is to show that the functions Δ_{FROM}, Δ_{WHERE}, Δ_{SELECT} capture the functionalities of the restricted S2R, R2R, and R2S operators (of the considered core setting of CQL), respectively.

- *S2R.* Regarding Δ_{FROM}, the translation passes on the parameters from the CQL window specifications to the window operators, which then will be passed to the corresponding window functions. By the definition of the window functions, tuples will be selected under the same principle by the window operators as in the `FROM` clause. This mimics the S2R operators that converts all input streams to relations at query time in CQL.
- *R2R.* Regarding Δ_{WHERE}, in case of comparing for equality of attributes with identical names, i.e., `P1.X=P2.X`, the comparison is implicitly reflected by the substitution for identical variables of the two respective predicates $p_1(X, \ldots)$ and $p_2(X, \ldots)$ at the step of grounding the translated schematic rule. For other comparisons, expressions of the forms (i) `P1.X1` *cmp* `P2.X2` and (ii) `P1.X1 = P2.X2` *op* `P3.X3`, the translation just rewrites the operators in another syntax and applies the same semantics.
  Using atoms from Δ_{FROM} and Δ_{WHERE} in the body of rules captures basic relational algebra operators such as selection, Cartesian product, and join. Moreover, Δ_{SELECT} allows to capture the projection operator of relational algebra.
- *R2S.* Given that
  - the same input is fed into a program $\Delta(Q)$ and a set of queries Q, and
  - the rules capture the operators from relational algebra,

  it follows that the body of a rule is satisfied iff the `WHERE` clause of the corresponding query is satisfied. Thus, the head of the rule is true in the (single) answer of $\Delta(Q)$ iff the corresponding query returns a respective result.
  Therefore, when we evaluate the program $\Delta(Q)$ at every time point at which the set of queries Q is triggered, the single answer at each time point corresponds to the results produced by the queries. This captures the R2S operator RStream. □