Towards Ideal Semantics for Analyzing Stream Reasoning

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

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 are 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 a first step towards an ideal semantics that allows for exact descriptions and comparisons of stream reasoning systems.

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 [4] studied by the database community, input tuples dynamically arrive at the processing systems in form of possibly infinite streams. To deal with unboundedness of data, such systems typically apply window operators to obtain snapshots of recent data. The user then runs continuous queries which are either periodically driven by time or eagerly driven by the arrival of new input. The Continuous Query Language (CQL) [3] is a well-known stream processing language. It has a syntax close to SQL 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 [16], i.e., logical reasoning on streaming data. To illustrate our contributions on this topic, we use an example from the public transport domain.

Example 1 To monitor a city’s public transportation, the city traffic center receives sensor data at every stop regarding tram/bus appearances of the form \( tr(X, P) \) and \( bus(X, P) \) where \( X, P \) hold the tram/bus and stop identifiers, respectively. On top of this streaming data tuples or atoms), one may ask different queries, e.g., to monitor the status of the public transport system. To keep things simple, we start with stream processing queries:

\[ q_1 \) At stop \( P \), did a tram and a bus arrive within the last \( 5 \) min?
\[ q_2 \) At stop \( P \), did a tram and a bus arrive at the same time within the last \( 5 \) min?

Consider the scenario of Fig. 1 which depicts arrival times of trams and buses. The answer to query \( q_2 \) is yes for stop \( p_2 \) and all time points from 2 to 7. Query \( q_1 \) also succeeds for \( p_1 \) from 11 to 13.

As for stream reasoning, later we will additionally consider a more involved query, where we are interested in whether a bus always arrived within three minutes after the last two arrivals of trams.

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 [14] 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) [9, 11] or extending Datalog to sequential logic programs [17]. 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, nonmonotonicity, or non-determinism. Such features are important to deal with missing of incomplete data, which can 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 [9] periodically calls the dlvhex solver [10] but is not capable of incremental reasoning and thus fails under heavy load of data. StreamLog [17] is an extension of Datalog towards stream reasoning. It always computes a single model and does not consider windows. Time-decaying logic programs [11] attempt to implement time-based windows in reactive ASP [13] but the relation to other stream processing/reasoning approaches has not yet been explored.

Moreover, as observed in [8], 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 [15], or informal examination on specific examples.
systematic investigation, however, requires a formalism to rigorously describe the expressivity and the properties of a language.

**Contributions.** We present a first step towards a formal framework for stream reasoning that (i) provides a common ground to express concepts from different stream processing/reasoning formalisms and engines; (ii) allows systematic analysis and comparison between existing stream processing/reasoning semantics; and (iii) also provides a basis for extension towards more expressive stream reasoning. Moreover, we present (iv) exemplary formalizations based on a running example, and (v) compare our approach to existing work.

Thereby, we aim at capturing idealized stream reasoning semantics where no information is dropped and semantics are characterized as providing an abstract view over the entire stream. Second, we idealize with respect to implementations and do not consider processing time, delays or outages in the semantics itself. Moreover, we allow for a high degree of expressivity regarding time reference: We distinguish notions of truth of a formula (i) at specific time points, (ii) some time point within a window, or (iii) all time points in a window. Moreover, we allow (iv) for nested window operators, which provide a means to reason over streams within the language itself (a formal counterpart to repeated runs of continuous queries).

## 2 Streams

In this section, we introduce a logic-oriented view of streams and formally define generalized versions of prominent window functions.

### 2.1 Streaming Data

A stream is usually seen as a sequence, set or bag of tuples with a timestamp. Here, we view streams as functions from a discrete time domain to sets of logical atoms and assume no fixed schema for tuples.

We build upon mutually disjoint sets of predicates $\mathcal{P}$, constants $\mathcal{C}$, variables $\mathcal{V}$ and time variables $\mathcal{U}$. The set $\mathcal{T}$ of terms is given by $\mathcal{C} \cup \mathcal{V}$ and the set $\mathcal{A}$ of atoms is defined as $\{p(t_1, \ldots, t_n) \mid p \in \mathcal{P}, t_1, \ldots, t_n \in \mathcal{T}\}$. The set $\mathcal{G}$ of ground atoms contains all atoms $p(t_1, \ldots, t_n) \in \mathcal{A}$ such that $\{t_1, \ldots, t_n\} \subseteq \mathcal{C}$. If $i, j \in \mathbb{N}$, the set $[i, j] = \{k \in \mathbb{N} \mid i \leq k \leq j\}$ is called an interval.

**Definition 1 (Stream)** Let $T$ be an interval and $v: \mathbb{N} \rightarrow 2^\mathcal{A}$ an interpretation function such that $v(t) = \emptyset$ for all $t \in \mathbb{N} \setminus T$. Then, the pair $S = (T, v)$ is called a stream, and $T$ is called the timeline of $S$.

The elements of a timeline are called time points or timestamps. A stream $S' = (T', v')$ is a substring or window of stream $S = (T, v)$, denoted $S' \subseteq S$, if $T' \subseteq T$ and $v'(t') \subseteq v(t)$ for all $t' \in T'$. The cardinality of $S$, denoted $\#S$, is defined by $\Sigma_{t \in T}[v(t)]$. The restriction of $S$ to $T' \subseteq T$, denoted $S|_{T'}$, is the stream $(T', v|_{T'})$, where $v|_{T'}$ is the usual domain restriction of function $v$.

**Example 2 (cont’d)** The input for the scenario in Example 1 can be modeled as a stream $S = (T, v)$ where $T = [0, 13]$ and

\[
\begin{align*}
v(2) &= \{\text{tr}(a, p_1), \text{bus}(c, p_1)\} \quad v(11) = \{\text{bus}(e, p_2)\} \\
v(8) &= \{\text{tr}(d, p_2)\} \quad v(t) = \emptyset \text{ otherwise.}
\end{align*}
\]

The interpretation $v$ can be equally represented as the following set:

\[
\{2 \rightarrow \{\text{tr}(a, p_1), \text{bus}(c, p_1)\}, 8 \rightarrow \{\text{tr}(d, p_2)\}, 11 \rightarrow \{\text{bus}(e, p_2)\}\}
\]

### 2.2 Windows

An essential aspect of stream reasoning is to limit the considered data to so-called windows, i.e., recent substreams, in order to limit the amount of data and forget outdated information.

**Figure 2.** Time-based window $w^2_{3,1}$ with range $(2, 1)$ and step size 3

**Definition 2 (Window function)** A window function maps from a stream $S = (T, v)$ and a time point $t \in T$ to a window $S' \subseteq S$.

The usual time-based window of size $\ell$ [3] contains only the tuples of the last $\ell$ time units. We give a generalized definition where the window can also include the tuples of the future $u$ time points. Based on query time $t$ and a step size $d$, we derive a pivot point $t'$ from which an interval $[t_\text{bus}, t_u]$ is selected by looking backward (resp., forward) $\ell$ (resp., $u$) time units from $t'$, i.e., $t_\text{bus} = t'$ and $t_u = t' + u = t_\text{bus}$.

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

\[
w_{t,u}^d(S, t) = (T', v|_{T'}),
\]

where $T' = [t_\text{bus}, t_u]$, $t_\text{bus} = \max\{t_{\text{min}}, t' - \ell\}$ with $t' = \lceil \frac{d}{u} \rceil \cdot d$, and $t_u = \min\{t' + u, t_{\text{max}}\}$.

For time-based windows that target only the past $\ell$ time points, we abbreviate $w_{t,0}^d$ with $w^d_t$. For windows which target only the future, we write $w_{0,u}^d$ for $w_{t,0}^d$. If the step size $d$ is omitted, we take $d = 1$. Thus, the standard sliding window with range $\ell$ is denoted by $w^\ell$.

The CQL [3] syntax for $w^\ell_t$ is [Range 1 Slide d] and $w^\ell$ corresponds to [Range 1]. Moreover, the window [Now] equals [Range 0] and thus corresponds to $w^0$. The entire past stream, selected by [Range Unbounded] in CQL, is obtained by $w^\ell$, where $t$ is the query time. To consider the entire stream (including the future), we can use $w^\infty$, where $n = \max T$.

Furthermore, we obtain tumbling windows by setting $d = \ell + u$.

**Example 3 (cont’d)** To formulate the monitoring over the stream $S$ of Example 2, one can use a time-based window $w^5_1$ with a range of 5 minutes (to the past) and step size of 1 minute, i.e., the granularity of $T$. The results of applying this window function at $t = 5, 11$ are

\[
w^5_1(S, 5) = ([0, 5], \{2 \rightarrow \{\text{tr}(a, p_1), \text{bus}(c, p_1)\}\}) \quad \text{and} \quad w^5_1(S, 11) = ([6, 11], \{8 \rightarrow \{\text{tr}(d, p_2)\}, 11 \rightarrow \{\text{bus}(e, p_2)\}\}).
\]

Moreover, consider a time-based (tumbling) window with range $(2, 1)$ and step size 3. For $t_1 = 5$, we have $t_1' = \lceil \frac{3}{2} \rceil = 3$, thus $T'_1 = [\max\{0, 3 - 2\}, \min\{3 + 1, 13\}] = [1, 4]$. For $t_2 = 11$, we get $t_2' = 9$ and $T'_2 = [7, 10]$. The windows for $t = 5, 11$ are

\[
w^{2,1}_{3,5}(S, 5) = ([1, 4], \{2 \rightarrow \{\text{tr}(a, p_1), \text{bus}(c, p_1)\}\}) \quad \text{and} \quad w^{2,1}_{3,5}(S, 11) = ([7, 10], \{8 \rightarrow \{\text{tr}(d, p_2)\}\}).
\]

Figure 2 illustrates the progression of this window with time.
The goal of the standard tuple-based window with count $n$ is to fetch the most recent $n$ tuples. Again, we give a more general definition which may consider future tuples. That is, relative to a time point $t \in T = [t_{\text{min}}, t_{\text{max}}]$, we want to obtain the most recent $\ell$ tuples (of the past) and next $u$ tuples in the future. Thus, we must return the stream restricted to the smallest interval $T' = [t_{\ell}, t_u] \subseteq T$, where $t_{\ell} \leq t \leq t_u$, such that $S$ contains $\ell$ tuples in the interval $[t_{\ell}, t]$ and $u$ tuples in the interval $[t + 1, t_u]$. In general, we have to discard tuples arbitrarily at time points $t_{\ell}$ and $t_u$ in order to receive exactly $\ell$ and $u$ tuples, respectively. In extreme cases, where fewer than $\ell$ tuples exist in $[t_{\text{min}}, t]$, respectively fewer than $u$ tuples in $[t + 1, t_{\text{max}}]$, we return all tuples of the allowed intervals. Given $t \in T$ and the tuple counts $\ell, u \in \mathbb{N}$, we define the tuple time bounds $t_{\ell}$ and $t_u$ as

\[ t_{\ell} = \max \{t_{\text{min}} \cup \{t' | \ t_{\text{min}} \leq t' \leq t \land \#(S|_{t_{\ell}+1}) \geq \ell \}, \]  
\[ t_u = \min \{t_{\text{max}} \cup \{t' | t + 1 \leq t' \leq t_{\text{max}} \land \#(S|_{t_{\ell}+1}) \geq u \}. \]

**Definition 4 (Tuple-based window)** Let $S = (T, v)$ be a stream and $t \in T$. Moreover, let $\ell, u \in \mathbb{N}$, $T_\ell = [t_{\ell}, t]$ and $T_u = [t + 1, t_u]$, where $t_{\ell}$ and $t_u$ are the tuple time bounds. The tuple-based window with counts $(\ell, u)$ of $S$ at time $t$ is defined by

\[ w_{\ell,u}^S(S, t) = (T', v'(\cdot)) \]  
where \[ T' = [t_{\ell}, t_u] \] and

\[ v'(t') = \begin{cases} 
    v(t') & \text{for all } t' \in T' \setminus \{t_{\ell}, t_u\}, \\
    X_{\ell} & \text{if } t' = t_{\ell} \text{ and } \#(S|_{t_{\ell}+1}) \leq \ell, \\
    X_{u} & \text{if } t' = t_u \text{ and } \#(S|_{t_{\ell}+1}) > \ell, \\
    t_{\ell} & \text{if } t' = t_{\ell} \text{ and } \#(S|_{t_{\ell}+1}) \leq \ell, \\
    t_u & \text{if } t' = t_u \text{ and } \#(S|_{t_{\ell}+1}) > \ell 
\end{cases}, \]  

where $X_{\ell} = (0, \ldots, 0)$, $X_{u} = (\ell + 1, \ldots, \ell + u)$ and $T' = [t_{\ell}, t_u]$. Note that this approach would simplify the definition of $w_{\ell,u}^S(S, t)$, requiring only to select $T' = [t_{\ell}, t_u]$. We abbreviate the usual tuple-based window operator $w_{\ell,u}^0$, which looks only into the past, by $w_{\ell}^0$. Similarly, $w_{\ell,u}^+ u$ stands for $w_{\ell,u}^{0}$.**

**Example 4 (cont’d)** To get the last 3 appearances of buses or trams from stream $S$ in Example 2 at time point 11, we can apply a tuple-based window with counts $(3,0)$, and obtain two possible windows $(T', v'_1)$ and $(T', v'_2)$, where $T' = [2, 11]$, and

\[ v'_1 = \{2 \rightarrow \{\text{tr}(a, p_1)\}, 8 \rightarrow \{\text{tr}(d, p_2)\}\}, \]
\[ v'_2 = \{2 \rightarrow \{\text{bus}(c, p_1)\}, 8 \rightarrow \{\text{tr}(d, p_2)\}\}, \]

The two interpretations differ at time point 2, where either $\text{tr}(a, p_1)$ or $\text{bus}(c, p_1)$ is picked to complete the collection of 3 tuples.

The CQL syntax for the tuple-based window is $[\text{Rows } n]$, which corresponds to $w_{\ell,u}^0$. Note that in CQL a single stream contains tuples of a fixed schema. In the logic-oriented view, this would translate to having only one predicate. Thus, applying a tuple-based window on a stream in our sense would amount to counting tuples across different streams. To enable counting of different predicates in separation, we introduce a general form of partition-based windows.

The partition-based window CQL applies a tuple-based window function on substrings which are determined by a sequence of attributes. The syntax $[\text{Partition By } a_1, \ldots, a_k \text{ Rows } N]$ means that tuples are grouped into substrings by identical values $a_1, \ldots, a_k$ of attributes $A_1, \ldots, A_k$. From each substring, the $N$ tuples with the largest timestamps are returned.

Here, we have no notion of attributes. Instead, we employ a general total index function $\text{idx}_i : \mathbb{G} \to I$ from ground atoms to a finite index set $I \subseteq \mathbb{N}$, where for each $i \in I$ we obtain from a stream $S = (T, v)$ a substring $\text{idx}_i(S) = (T, v_i)$ by taking $v_i(t) = \{a \in v(t) | \text{idx}(a) = i\}$. Moreover, we allow for individual tuple counts $n(i) = \langle t_{\ell}(i), t_u(i) \rangle$ for each substring $S_i$.

**Definition 5 (Partition-based window)** Let $S = (T, v)$ be a stream, $\text{idx}_i : \mathbb{G} \to I \subseteq \mathbb{N}$, an index function, and for all $i \in I$ let $n(i) = \langle t_{\ell}(i), t_u(i) \rangle \in \mathbb{N} \times \mathbb{N}$ and $S_i = \text{idx}_i(S)$. Moreover, let $t \in T$ and $w_{\ell,u}^{n(i)}(S_i, t) = ((t'_{\ell}(i), t'_{u}(i)), v'_i)$ be the tuple-based window of counts $(\ell(i), u(i))$ of $S_i$, at time $t$. Then, the partition-based window of counts $\langle t_{\ell}(i), t_u(i) \rangle \in I$ at time $t$ relative to $\text{idx}_i$ is defined by

\[ w_{\ell,u}^{n(i)}(S, t) = (T', v'(\cdot)), \]  
where $T' = [\min t'_{\ell}, \max t'_{u}]$ and $v'(t') = \bigcup_{i \in I} v_i(t'(i))$ for all $t' \in T'$.

Note that, in contrast to schema-based stream reasoning, we have multiple kinds of tuples (predicates) in one stream. Whereas other approaches may use tuple-based windows of different counts on separate streams, we can have separate tuple-counts on the corresponding substrings of a partition-based window on a single stream.

**Example 5 (cont’d)** Suppose we are interested in the arrival times of the last 2 trams, but we are not interested in buses. To this end, we construct a partition-based window $w_{\ell,u}^{n(1)}$ as follows. We use index set $I = \{1, 2\}$, and $\text{idx}_i(\rho(X, Y)) = 1$ iff $p = \text{tr}$. For the counts in the tuple-based windows of the substrings, we use $n(1) = (2, 0)$ and $n(2) = (0, 0)$. We obtain the substrings

\[ S_1 = ([2, 13], \{2 \rightarrow \{\text{tr}(a, p_1)\}, 8 \rightarrow \{\text{tr}(d, p_2)\}\}), \]  
\[ S_2 = ([2, 13], \{2 \rightarrow \{\text{bus}(c, p_1)\}, 11 \rightarrow \{\text{bus}(e, p_2)\}\}). \]

and the respective tuple-based windows

\[ w_{\ell,u}^{n(1)}(S_1, 13) = ([2, 13], \{2 \rightarrow \{\text{tr}(a, p_1)\}, 8 \rightarrow \{\text{tr}(d, p_2)\}\}), \]
\[ w_{\ell,u}^{n(2)}(S_2, 13) = ([13, 13], \emptyset). \]

Consequently, we get $w_{\ell,u}^{n(1)}(S, 13) = ([2, 13], v'_1)$, where $v'_1$ is

\[ 2 \rightarrow \{\text{tr}(a, p_1)\}, 8 \rightarrow \{\text{tr}(d, p_2)\}. \]

3 Reasoning over Streams

We are now going to utilize the above definitions of streams and windows to formalize a semantics for stream reasoning.

3.1 Stream Semantics

Towards rich expressiveness, we provide different means to relate logical truth to time. Similarly as in modal logic, we will use operators $\square$ 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 $\Diamond$ to refer to specific time points. To obtain a window of the stream, we employ window operators $\llbracket \cdot \rrbracket$. 

We say a formula $\alpha$ is ground, if all its atoms are ground and for all occurrences of form $\alpha_1, \alpha_2$ in $\alpha$ it holds that $\alpha_1, \alpha_2 \in \mathcal{N}$. In the following semantics definition, we will consider the input stream (astream) which remains unchanged, as well as dynamic substreams thereof which are obtained by (possibly nested) applications of window functions. To this end, we define a stream choice to be a function that returns a stream based on two input streams. Two straightforward stream choices are $ch_i$, for $i \in \{1, 2\}$, defined by $ch_i(s_1, s_2) = s_i$. Given a stream choice $ch$, we obtain for any window function $w$ an extended window function $\hat{w}$ by $\hat{w}(s_1, s_2, t) = w(ch(s_1, s_2), t)$ for all $t \in \mathcal{N}$. We say $\hat{w}$ is the extension of $w$ due to $ch$.

Definition 7 (Structure) Let $S_M = (T, v)$ be a stream, $I \subseteq \mathcal{N}$ a finite index set and let $W$ be a function that maps every $i \in I$ to an extended window function. The triple $M = (T, v, W)$ is called a structure and $S_M$ is called the up-stream of $M$.

We now define when a ground formula holds in a structure.

Definition 8 (Entailment) Let $M = (T, v, W)$ be a structure. For a sub-stream $S = (T_s, v_s)$ of $(T, v)$, we define the entailment relation $\models$ between $(M, S, t)$ and formulas. Let $t \in T$, $\alpha, \beta \in \mathcal{F}_k$ be ground formulas and let $\hat{w} = W(i)$. Then,

\[
M, S, t \models \alpha \quad \text{iff} \quad \alpha \in v_s(t),
\]
\[
M, S, t \models \neg \alpha \quad \text{iff} \quad M, S, t \not\models \alpha,
\]
\[
M, S, t \models \alpha \land \beta \quad \text{iff} \quad M, S, t \models \alpha \text{ and } M, S, t \models \beta,
\]
\[
M, S, t \models \alpha \lor \beta \quad \text{iff} \quad M, S, t \models \alpha \text{ or } M, S, t \models \beta,
\]
\[
M, S, t \models \alpha \rightarrow \beta \quad \text{iff} \quad M, S, t \not\models \alpha \text{ or } M, S, t \models \beta,
\]
\[
M, S, t \models \Box \alpha \quad \text{iff} \quad M, S, t \models \alpha \text{ for some } t' \in T_s,
\]
\[
M, S, t \models \Diamond \alpha \quad \text{iff} \quad M, S, t' \models \alpha \text{ for all } t' \in T_s,
\]
\[
M, S, t \models \Box \exists \alpha \quad \text{iff} \quad M, S, t' \models \exists \alpha \text{ and } t' \in T_s,
\]
\[
M, S, t \models \exists \Box \alpha \quad \text{iff} \quad M, S', t \models \alpha \text{ where } S' = \hat{w}(S_M, S, t).
\]

If $M, S, t \models \alpha$ holds, we say $(M, S, t)$ entails $\alpha$. Intuitively, $M$ contains the up-stream $S_M$ which remains unchanged and $S$ is the currently considered window. An application of a window operator $\oplus$ utilizes the extended window $\hat{W}(i)$ which can take into account both the up-stream $S_M$ and the current window $S$ to obtain a new view, as we will discuss later. The operators $\otimes$ and $\circ$ are used to evaluate whether a formula holds at some time point, respectively at all time points in the timeline $T_s$ of $S$. The operator $\circ$ allows to evaluate whether a formula holds at a specific time point $t$ in $S$.

Example 6 (cont’d) Let $M = (T, v, W)$, where $S_M = (T, v)$ is the stream $S$ from Example 2 and $\hat{W}(1) = w^0$, i.e., the extension of $w^0$ of Example 3 due to $ch_2$. Consider the following formula:

\[
\alpha = \Box_2 (\Diamond 2 (\exists \text{tr}(d, p_2) \land \exists \text{bus}(e, p_2))
\]

We verify that $M, S_M, 11 \models \alpha$ holds. First, the window operator $\oplus \otimes \exists \text{tr}(d, p_2)$ selects the substream $(T_s', v')$, where $T_s' = [6, 11]$ and $v' = \{v'_i \} = \{8 \rightarrow \{\text{tr}(d, p_2)\}, 11 \rightarrow \{\text{bus}(e, p_2)\}\}$. Next, to see that $(M, S', 11)$ entails $\exists \text{tr}(d, p_2) \land \exists \text{bus}(e, p_2)$, we have to find time points in the timeline $T_s'$ of the current window $S'$, such that $\text{tr}(d, p_2)$ and $\exists \text{bus}(e, p_2)$ hold, respectively. Indeed, for $8$ and $11$, we have $M, S_1, 8 \models \exists \text{tr}(d, p_2)$ and $M, S_1, 11 \models \exists \text{bus}(e, p_2)$.

Example 7 (cont’d) We formalize the queries of Ex. 1 as follows:

\[
q_1 = \Box I (\otimes \text{tr}(X, P) \land \exists \text{bus}(Y, P))[u]
\]
\[
q_2 = \Box I (\otimes \text{tr}(X, P) \land \exists \text{bus}(Y, P))[u]
\]

The query $q = \Box I (\otimes (\text{tr}(a, p_1) \land \exists \text{bus}(c, p_1)))$ is ground iff $t \in [2, 7]$. The evaluation of $q$ on stream $M, S_M, t \models q$ holds, and $q = \Box I (\otimes (\text{tr}(a, p_1) \land \exists \text{bus}(c, p_1)))$ for all $t \in [2, 7]$ and $M, S_M, t \models \Box I (\otimes (\text{tr}(d, p_2) \land \exists \text{bus}(e, p_2)))$ for all $t \in [11, 13]$.

Thus, the following set of substitutions is the answer to $q_1$ in $M$:

\[
\{t \rightarrow a, Y \rightarrow c, P \rightarrow p_1, \{u \rightarrow t\}\} \quad \text{for } t \in [2, 7] \cup \{t \rightarrow d, Y \rightarrow c, P \rightarrow p_2, \{u \rightarrow t\}\} \quad \text{for } t \in [11, 13]
\]

Exact time reference. With the operator $\circ$, we can ask whether a formula holds at a specific time point $t$. In its non-ground version, we can utilize this operator for the selection of time points.

Example 8 (cont’d) Let $\alpha = \text{tr}_\text{am}(X, P) \land \exists \text{bus}(Y, P)$. For each of the queries $\oplus \circ \exists \alpha[13]$ and $\exists \alpha[U]$, the time assignments for $U$ in the answers will map to time points when a tram and a bus arrived.
simultaneously at the same stop. In both cases, the single answer is $\{(X \rightarrow a, Y \rightarrow c, P \rightarrow p_1), (U \rightarrow 2)\}$. Note that omitting the operator $\alpha$ in the first query would give an empty answer, since the subformula $\alpha$ does not hold at time point 13.

We observe that the operator $\alpha$ allows to replay a historic query. At any time $t' > t$, we can ask $\alpha[t][t']$ to simulate a previous query $\alpha[t]$.

**Nested windows.** Typically, window functions are used exclusively to restrict the processing of streams to a recent subset of the input. In our view, window functions provide a flexible means to reason over temporally local contexts within larger windows. For these nested windows we carry both $M$ and $S$ for the entailment relation.

**Example 9 (cont’d)** Consider the following additional query ($q_3$): At which stops $P$, for the last 2 two trams $X$, did a bus $Y$ arrive within 3 minutes? To answer ($q_3$) at time point 13, we ask

$$q_3 = \exists t \exists (X, P) \rightarrow \exists t' \exists bus(Y, P)[13].$$

For $\exists t$, we can use the extension $\mathbb{w}^{\exists t}_n$ of the partition-based window $\mathbb{w}^{\exists t}_n$ of Example 5. Applying $\hat{W}(1)$ on the stream $S = (T, v)$ in the previous examples yields $S' = (T', v')$, where $T' = [2, 13]$ and $v' = \{(t \rightarrow (tr(a, p_1)), 8 \rightarrow (tr(d, p_2)))\}$. That is, after applying this window, the current window $S'$ no longer contains information on buses. Consequently, to check whether a bus came in both cases within 3 minutes, we must use the upstream $S_M$. Thus, the second extended window $\hat{W}(2) = \hat{w}^{W_3}$ is the extension of the time-based window $w^{W_3}$, which looks 3 minutes into the future, due to the stream choice $c_1$. Hence, $\hat{w}^{W_3}$ will create a window based on $S_M$ and not on $S'$. The two time points in $T'$ where a tram appears are 2 and 8, with $p_1$ and $p_2$, respectively. Applying $\hat{W}(2)$ there yields the streams $S'_M = (T'_M, v'_M)$ and $S''_M = (T''_M, v''_M)$, where

$$T''_M = [2, 5], \quad v''_M = \{2 \rightarrow (tr(a, p_1), bus(c, p_1))\}, \quad \text{and} \quad T''_M = [8, 11], \quad v''_M = \{8 \rightarrow (tr(d, p_2)), 11 \rightarrow (bus(e, p_2))\}.$$ 

In both streams, we find a time point with an atom $bus(Y, p_1)$ with the same stop $p_1$ as the tram. Thus, in both cases the subformula $\hat{w}^{bus}(Y, P)$ is satisfied and so the implication $tr(X, P) \rightarrow \exists t_2 \exists bus(Y, P)$ holds at every point in time of the stream selected by $\exists t$. Hence, the answer to the query is

$$q_3 = \{(X \rightarrow a, Y \rightarrow c, P \rightarrow p_1, \emptyset), \{(X \rightarrow d, Y \rightarrow e, P \rightarrow p_2, \emptyset)\}.$$ 

### 4 Discussion and Related Work

In this section we discuss the relationship of this ongoing work with existing approaches from different communities.

**Modal logic.** The presented formalism employs operators $\exists$ and $\forall$ as in modal logic [6]. Also, the definition of entailment uses a structure similar to Kripke models for multi-modal logics. However, instead of static accessibility relations, we use window functions which take into account not only the worlds (i.e., the time points) but also the interpretation function. To our best knowledge, window operators have been considered neither in modal logics nor temporal logics.

**CQL.** By extending SQL to deal with input streams, CQL queries are evaluated based on three sets of operators:

(i) **Stream-to-relation** operators apply function windows 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** operators allow for modification of relations similarly as in relational algebra, respectively SQL.

(iii) **Relation-to-stream** 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.

The proposed semantics has means to capture these operators:

(i) The window operators $\exists t$, keep the timestamps of the selected atoms, whereas the stream-to-relation operator discards them. The CQL query for tuple $x$ thus corresponds to a query $\exists t_2$ of the present setting. A stream in CQL belongs to a fixed schema. As noted earlier, this corresponds to the special case with only one predicate. CQL’s partition-based window is a generalization of the tuple-based window defined there. In turn, the presented partition-based window generalizes the one of CQL.

(ii) Some relational operators can essentially be captured by logical connectives, e.g., the join by conjunction. Some operators like projection will require an extension of the formalism towards rules. Moreover, we did not consider arithmetic operators and aggregation functions, which CQL inherits from SQL.

(iii) The answer to a non-ground query $\alpha[u]$ is a set of query assignments $(\sigma, \tau)$. To capture the RStream of CQL, we can group these assignments by the time variable $u$.

**Example 10** Queries ($q_1$) and ($q_2$) from Example 1 can be expressed in CQL. We assume that both streams have the attributes $X$ and $P$, corresponding to the first, respectively second argument of predicates $tr$ and $bus$. For ($q_1$), we can use:

```sql
SELECT * FROM tr [RANGE 5], bus [RANGE 5] WHERE tr.P = bus.P
```

On the other hand, ($q_2$) needs two CQL queries.

```sql
SELECT AS tr_bus FROM tr [NOW], bus [NOW] WHERE tr.P = bus.P
SELECT FROM tr_bus [RANGE 5]
```

Here, the first query produces a new stream that contains only simultaneous tuples and the second one covers the range of 5 minutes.

Traditionally, stream reasoning approaches use **continuous queries**, i.e., repeated runs of queries with snapshot semantics to deal with changing information. In this work, we go a step further and enable reasoning over streams within the formalism itself by means of nested windows. One can only mimic this feature with CQL’s snapshot semantics when timestamps are part of the schema and explicitly encoded. Likewise, queries to future time points can be emulated in this way, as the next example shows.

**Example 11 (cont’d)** In Example 9, we considered bus arrivals within 3 minutes after the last 2 trams. In CQL, such a query is not possible on the assumed schema. However, by adding a third attribute $TS$ that carries the timestamps to the schema, the following CQL query yields the same results.

```sql
SELECT * FROM tr [ROWS 2],
```

Note that we need no partition-based window here, since trams and buses arrive from different input streams. Moreover, we must use the unbounded window for buses to cover nesting of windows in ($q_1$) because windows in CQL are applied at query time and not the time where a tram appearance is notified.
Furthermore, nested CQL queries and aggregation inherited from SQL are promising to mimic the behavior of operator ⊓. With according rewriting, CQL engines like STREAM [2] could be used to realize the proposed semantics.

SECRET. In [8] a model called SECRET is proposed to analyze the execution behavior of different stream processing engines (SPEs) from a practical point of view. The authors found that even the outcome of identical, simple queries vary significantly due to the different underlying processing models. There, the focus is on understanding, comparing and predicting the behaviour of engines. In contrast, we want to provide means that allow for a similar analytical study for the semantics of stream reasoning formalisms and engines. The two approaches are thus orthogonal and can be used together to compare stream reasoning engines based on different input feeding modes as well as different reasoning expressiveness.

Reactive ASP. The most work related to expressive stream reasoning with rules [11] is based on Reactive ASP [12]. This setting introduces logic programs that extend over time. Such programs have the following components. Two components \( P \) and \( Q \) are parametrized with a natural number \( t \) for time points. In addition, a basic component \( B \) encodes background knowledge that is not time-dependent. Moreover, sequences of pairs of arbitrary logic programs \( (E_i,F_i) \), called online progression are used. While \( P \) and \( E_i \) capture accumulated knowledge, \( Q \) and \( F_i \) are only valid at specific time points. Compared to reactive ASP, our semantics has no mechanism for accumulating programs, and we take only streams of atoms/facts, but no background theories. Therefore, a framework based on idealized semantics with extension to rules should be able to capture a fragment of reactive ASP where \( P \) and \( F_i \) are empty and \( E_i \) contains only facts. The foreseeable conversion can be as follows: convert rules in \( Q \) by applying an unbounded window on all body atoms of a rule, using \( \theta_i \) to query the truth value of the atoms at time point \( t \). Then, conclude the head to be true at \( t \) and feed facts from \( E_i \) to the input stream \( S \).

StreamLog. Another logic-based approach towards stream reasoning is StreamLog [17]. It makes use of Datalog and introduces temporal predicates whose first arguments are timestamps. By introducing sequential programs which have syntactical restrictions on temporal rules, StreamLog defines non-blocking negation (for which Closed World Assumption can be safely applied) that can be used in recursive programs in a stream setting. Since sequential programs are locally stratified, they have efficiently computable perfect (i.e., unique) models. Similar to capturing a fragment of Reactive ASP, we can capture StreamLog by converting temporal atoms \( p(t, x_1, \ldots, x_n) \) to expressions \( \Theta(p(x_1, \ldots, x_n)) \) and employing safety conditions to rules to simulate non-blocking negation. Moreover, we plan for having weaker notions of negation that might block rules but just for a bounded number of time points to the future.

ETALIS. The ETALIS system [1] aims at adding expressiveness to Complex Event Processing (CEP). It provides a rule-based language for pattern matching over event streams with declarative monotonic semantics. Simultaneous events are not allowed and windows are not regarded as first-class objects in the semantics, but they are available at the system implementation level. Tuple-based windows are also not directly supported. Furthermore, nesting of windows is not possible within the language, but it can be emulated with multiple rules as in CQL. On the other hand, ETALIS models complex events with time intervals and has operators to express temporal relationships between events.

5 Conclusion

We presented a first step towards a theoretical foundation for (idealistc) semantics of stream reasoning formalisms. Analytical tools to characterize, study and compare logical aspects of stream engines have been missing. To fill this gap, we provide a framework to reason over streaming data with a fine-grained control over relating the truth of tuples with their occurrences in time. It thus, e.g., allows to capture various kinds of window applications on data streams. We discussed the relationship of the proposed formalism with existing approaches, namely CQL, SECRET, Reactive ASP, StreamLog, and ETALIS.

Next steps include extensions of the framework to formally capture fragments of existing approaches. Towards more advanced reasoning features like recursion and non-monotonicity, we aim at a rule-based semantics on top of the presented core. Furthermore, considering intervals of time as references is an interesting research issue. To improve practicality (as a tool for formal and experimental analysis) one might also develop an operational characterization of the framework. In a longer perspective, along the same lines with [7], we aim at a formalism for stream reasoning in distributed settings across heterogeneous nodes having potentially different logical capabilities.

REFERENCES