Towards Ideal Semantics for Analyzing Stream Reasoning[1]

Harald Beck and **Minh Dao-Tran** and **Thomas Eiter** and **Michael Fink** [2]

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. ∎

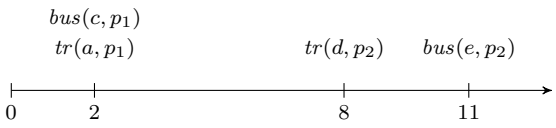

Figure 1. Traffic scenario with arrivals of trams and buses

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. A

[1] Supported by the Austrian Science Fund (FWF) project P26471.
[2] Institut für Informationssysteme, Technische Universität Wien. email: {beck,dao,eiter,fink}@kr.tuwien.ac.at

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 $\upsilon \colon \mathbb{N} \to 2^{\mathcal{G}}$ an* interpretation function *such that $\upsilon(t) = \emptyset$ for all $t \in \mathbb{N} \setminus T$. Then, the pair $S = (T, \upsilon)$ 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 $S' = (T', \upsilon')$ is a *substream* or *window* of stream $S = (T, \upsilon)$, denoted $S' \subseteq S$, if $T' \subseteq T$ and $\upsilon'(t') \subseteq \upsilon(t')$ for all $t' \in T'$. The *cardinality* of S, denoted $\#S$, is defined by $\Sigma_{t \in T} |\upsilon(t)|$. The *restriction of S to $T' \subseteq T$*, denoted $S|_{T'}$, is the stream $(T', \upsilon|_{T'})$, where $\upsilon|_{T'}$ is the usual domain restriction of function υ.

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

$$\begin{array}{lll} \upsilon(2) = \{tr(a, p_1), bus(c, p_1)\} & & \upsilon(11) = \{bus(e, p_2)\} \\ \upsilon(8) = \{tr(d, p_2)\} & & \upsilon(t) = \emptyset \text{ otherwise.} \end{array}$$

The interpretation υ can be equally represented as the following set: $\{2 \mapsto \{tr(a, p_1), bus(c, p_1)\}, 8 \mapsto \{tr(d, p_2)\}, 11 \mapsto \{bus(e, p_2)\}\}$ ∎

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.

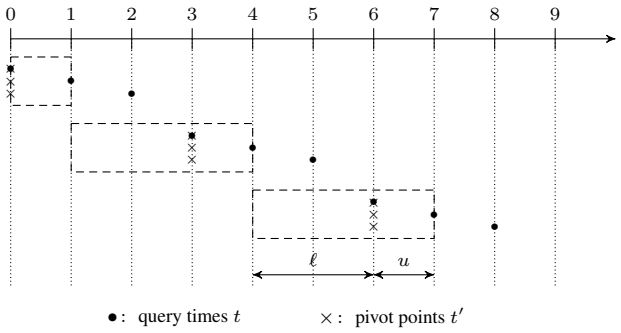

Figure 2. Time-based window $w_3^{2,1}$ with range $(2, 1)$ and step size 3

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

The usual time-based window of size ℓ [3] contains only the tuples of the last ℓ 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 $[t_\ell, t_u]$ is selected by looking backward (resp., forward) ℓ (resp., u) time units from t', i.e., $t_\ell + \ell = t'$ and $t' + u = t_u$.

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

$$w_d^{\ell,u}(S, t) = (T', \upsilon|_{T'}),$$

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

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

The CQL [3] syntax for w_d^ℓ is `[Range l Slide d]` and w^ℓ corresponds to `[Range l]`. 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^t, where t is the query time. To consider the entire stream (including the future), we can use w^n, 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 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

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

Moreover, consider a time-based (tumbling) window with range $(2, 1)$ and step size 3. For $t_1 = 5$, we have $t_1' = \lfloor \frac{5}{3} \rfloor \cdot 3 = 3$, 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

$$w_3^{2,1}(S, 5) = ([1, 4], \{2 \mapsto \{tr(a, p_1), bus(c, p_1)\}\}), \text{ and}$$
$$w_3^{2,1}(S, 11) = ([7, 10], \{8 \mapsto \{tr(d, p_2))\}\}).$$

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 $t \in T = [t_{min}, t_{max}]$, we want to obtain the most recent ℓ 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 ℓ 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_ℓ and t_u in order to receive *exactly* ℓ and u tuples, respectively. In extreme cases, where fewer than ℓ tuples exist in $[t_{min}, t]$, respectively fewer than u tuples in $[t+1, t_{max}]$, we return all tuples of the according intervals. Given $t \in T$ and the tuple counts $\ell, u \in \mathbb{N}$, we define the *tuple time bounds* t_ℓ and t_u as

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

Definition 4 (Tuple-based window) *Let $S = (T, \upsilon)$ 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_ℓ and t_u are the tuple time bounds. The* tuple-based window with counts (ℓ, u) of S at time t *is defined by*

$$w^{\#\ell,u}(S,t) = (T', \upsilon'|_{T'}), \text{ where } T' = [t_\ell, t_u], \text{ and}$$

$$\upsilon'(t') = \begin{cases} \upsilon(t') & \text{for all } t' \in T' \setminus \{t_\ell, t_u\} \\ \upsilon(t') & \text{if } t' = t_\ell \text{ and } \#(S|_{T_\ell}) \leq \ell \\ X_\ell & \text{if } t' = t_\ell \text{ and } \#(S|_{T_\ell}) > \ell \\ \upsilon(t') & \text{if } t' = t_u \text{ and } \#(S|_{T_u}) \leq u \\ X_u & \text{if } t' = t_u \text{ and } \#(S|_{T_u}) > u \end{cases}$$

where $X_q \subseteq \upsilon(t_q)$, $q \in \{\ell, u\}$, such that $\#(T_q, \upsilon'|_{T_q}) = q$.

Note that the tuple-based window is unique only if for both $q \in \{\ell, u\}$, $\upsilon'(t_q) = \upsilon(t_q)$, i.e., if all atoms at the endpoints of the selected interval are retained. There are two natural possibilities to enforce the uniqueness of a tuple-based window. First, if there is a total order over all atoms, one can give a deterministic definition of the sets X_q in Def. 4. Second, one may omit the requirement that *exactly* ℓ tuples of the past, resp. u tuples of the future are contained in the window, but instead demand the substream obtained by the smallest interval $[t_\ell, t_u]$ containing *at least* ℓ past and u future tuples. Note that this approach would simplify the definition to $w^{\#\ell,u}(S,t) = (T', \upsilon|_{T'})$, requiring only to select $T' = [t_\ell, t_u]$. We abbreviate the usual tuple-based window operator $w^{\#\ell,0}$, which looks only into the past, by $w^{\#\ell}$. Similarly, $w^{\#+u}$ stands for $w^{\#0,u}$.

Example 4 (cont'd) To get the last 3 appearances of trams or buses from stream S in Example 2 at time point 11, we can apply a tuple-based window with counts $(3, 0)$. The application $w^{\#3}(S, 11)$ can lead to two possible windows (T', υ'_1) and (T', υ'_2), where $T' = [2, 11]$, and

$$\upsilon'_1 = \{2 \mapsto \{tr(a, p_1)\}, 8 \mapsto \{tr(d, p_2)\}, 11 \mapsto \{bus(e, p_2)\}\},$$
$$\upsilon'_2 = \{2 \mapsto \{bus(c, p_1)\}, 8 \mapsto \{tr(d, p_2)\}, 11 \mapsto \{bus(e, p_2)\}\}.$$

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

The CQL syntax for the tuple-based window is `[Rows n]`, which corresponds to $w^{\#n}$. 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 substreams which are determined by a sequence of attributes. The syntax `[Partition By A1,...,Ak Rows N]` means that tuples are grouped into substreams by identical values $a_1, \ldots, a_k$ of attributes `A1`,..., `Ak`. From each substream, the `N` tuples with the largest timestamps are returned.

Here, we have no notion of attributes. Instead, we employ a general total *index function* $\mathrm{idx} : \mathcal{G} \to I$ from ground atoms to a finite *index set* $I \subseteq \mathbb{N}$, where for each $i \in I$ we obtain from a stream $S = (T, \upsilon)$ a substream $\mathrm{idx}_i(S) = (T, \upsilon_i)$ by taking $\upsilon_i(t) = \{a \in \upsilon(t) \mid \mathrm{idx}(a) = i\}$. Moreover, we allow for individual tuple counts $n(i) = (\ell_i, u_i)$ for each substream S_i.

Definition 5 (Partition-based window) *Let $S = (T, \upsilon)$ be a stream, $\mathrm{idx} : \mathcal{G} \to I \subseteq \mathbb{N}$, an index function, and for all $i \in I$ let $n(i) = (\ell_i, u_i) \in \mathbb{N} \times \mathbb{N}$ and $S_i = \mathrm{idx}_i(S)$. Moreover, let $t \in T$ and $w^{\#\ell_i,u_i}(S_i, t) = ([t_i^\ell, t_i^u], \upsilon'_i)$ be the tuple-based window of counts (ℓ_i, u_i) of S_i at time t. Then, the* partition-based window of counts $\{(\ell_i, u_i)\}_{i \in I}$ of S at time t relative to idx *is defined by*

$$w^{\#n}_{\mathrm{idx}}(S,t) = (T', \upsilon'), \text{ where } T' = [\min_{i \in I} t_i^\ell, \max_{i \in I} t_i^u],$$

and $\upsilon'(t') = \bigcup_{i \in I} \upsilon'_i(t')$ for all $t' \in T'$.

Note that, in contrast to schema-based streaming approaches, 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 substreams 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 $w^{\#n}_{\mathrm{idx}}$ as follows. We use index set $I = \{1, 2\}$, and $\mathrm{idx}(p(X, Y)) = 1$ iff $p = tr$. For the counts in the tuple-based windows of the substreams, we use $n(1) = (2, 0)$ and $n(2) = (0, 0)$. We obtain the substreams

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

and the respective tuple-based windows

$$w^{\#2}(S_1, 13) = ([2, 13], \{2 \mapsto \{tr(a, p_1)\}, 8 \mapsto \{tr(d, p_2)\}\}), \text{ and}$$
$$w^{\#0}(S_2, 13) = ([13, 13], \emptyset).$$

Consequently, we get $w^{\#n}_{\mathrm{idx}}(S, 13) = ([2, 13], \upsilon')$, where υ' is

$$\{2 \mapsto \{tr(a, p_1)\}, 8 \mapsto \{tr(d, p_2)\}.$$ ∎

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 $\Box$ 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. To obtain a window of the stream, we employ *window operators* $\boxplus_i$.

Definition 6 (Formulas $\mathcal{F}_k$) *The set $\mathcal{F}_k$ of* formulas (for k modalities) *is defined by the grammar*

$$\alpha ::= a \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \alpha \to \alpha \mid \Diamond\alpha \mid \Box\alpha \mid @_t\alpha \mid \boxplus_i\alpha$$

where a is any atom in $\mathcal{A}$, $i \in \{1, \dots k\}$, and $t \in \mathbb{N} \cup \mathcal{U}$.

We say a formula α is *ground*, if all its atoms are ground and for all occurrences of form $@_t\beta$ in α it holds that $t \in \mathbb{N}$. In the following semantics definition, we will consider the input stream (*urstream*) 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 $t \in \mathbb{N}$. We say $\hat{w}$ is the *extension of w (due to ch)*.

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

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

Definition 8 (Entailment) *Let $M = \langle T, \upsilon, \hat{W}\rangle$ be a structure. For a substream $S = (T_S, \upsilon_S)$ of (T, υ), we define the* entailment *relation $\Vdash$ between (M, S, t) and formulas. Let $t \in T$, $a \in \mathcal{G}$, and $\alpha, \beta \in \mathcal{F}_k$ be ground formulas and let $\hat{w}_i = \hat{W}(i)$. Then,*

$$\begin{array}{lll}
M,S,t \Vdash a & \textit{iff} & a \in \upsilon_S(t)\,,\\
M,S,t \Vdash \neg\alpha & \textit{iff} & M,S,t \nVdash \alpha,\\
M,S,t \Vdash \alpha \wedge \beta & \textit{iff} & M,S,t \Vdash \alpha \textit{ and } M,S,t \Vdash \beta,\\
M,S,t \Vdash \alpha \vee \beta & \textit{iff} & M,S,t \Vdash \alpha \textit{ or } M,S,t \Vdash \beta,\\
M,S,t \Vdash \alpha \to \beta & \textit{iff} & M,S,t \nVdash \alpha \textit{ or } M,S,t \Vdash \beta,\\
M,S,t \Vdash \Diamond\alpha & \textit{iff} & M,S,t' \Vdash \alpha \textit{ for some } t' \in T_S,\\
M,S,t \Vdash \Box\alpha & \textit{iff} & M,S,t' \Vdash \alpha \textit{ for all } t' \in T_S\,,\\
M,S,t \Vdash @_{t'}\alpha & \textit{iff} & M,S,t' \Vdash \alpha \textit{ and } t' \in T_S\,,\\
M,S,t \Vdash \boxplus_i\alpha & \textit{iff} & M,S',t \Vdash \alpha \textit{ where } S' = \hat{w}_i(S_M, S, t).
\end{array}$$

If $M, S, t \Vdash \alpha$ holds, we say (M, S, t) *entails* α. Intuitively, M contains the urstream S_M which remains unchanged and S is the currently considered window. An application of a window operator $\boxplus_i$ utilizes the extended window $\hat{W}(i)$ which can take into account both the urstream S_M and the current window S to obtain a new view, as we will discuss later. The operators $\Diamond$ and $\Box$ 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 $@_t$ allows to evaluate whether a formula holds at a specific time point t in T_S.

Example 6 (cont'd) Let $M = \langle T, \upsilon, \hat{W}\rangle$, where $S_M = (T, \upsilon)$ is the stream S from Example 2 and $\hat{W}(1) = \hat{w}^5$, i.e., the extension of w^5 of Example 3 due to ch_2. Consider the following formula:

$$\alpha = \boxplus_1(\Diamond tr(d, p_2) \wedge \Diamond bus(e, p_2))$$

We verify that $M, S_M, 11 \Vdash \alpha$ holds. First, the window operator $\boxplus_1$ selects the substream $S' = (T_{S'}, \upsilon')$, where $T_{S'} = [6, 11]$ and $\upsilon' = \upsilon|_{T'} = \{8 \mapsto \{tr(d, p_2)\}, 11 \mapsto \{bus(e, p_2)\}\}$. Next, to see that $(M, S', 11)$ entails $\Diamond tr(d, p_2) \wedge \Diamond bus(e, p_2)$, we have to find time points in the timeline $T_{S'}$ of the current window S', such that $tr(d, p_2)$ and $bus(e, p_2)$ hold, respectively. Indeed, for 8 and 11, we have $M, S_1, 8 \Vdash tr(d, p_2)$ and $M, S_1, 11 \Vdash bus(e, p_2)$. ∎

DEFINITION		SCOPE
$\Theta(t)$	$= t$	time points $t \in \mathbb{N}$
$\Theta(u)$	$= \tau(u)$	time variables $u \in \mathcal{U}$
$\Theta(c)$	$= c$	constants $c \in \mathcal{C}$
$\Theta(v)$	$= \sigma(v)$	variables $v \in \mathcal{V}$
$\Theta(p(t_1, \dots, t_n)) =$ $p(\Theta(t_1), \dots, \Theta(t_n))$		predicates $p \in \mathcal{P}$ and terms $t_i \in \mathcal{T}$
$\Theta(\alpha\,\mathbf{b}\,\beta)) = \Theta(\alpha)\,\mathbf{b}\,\Theta(\beta)$		$\mathbf{b} \in \{\wedge, \vee, \to\}$
$\Theta(\mathbf{u}\alpha)$	$= \mathbf{u}\,\Theta(\alpha)$	$\mathbf{u} \in \{\neg, \Diamond, \Box\} \cup \{\boxplus_i\}_{i\in\mathbb{N}}$
$\Theta(@_u\alpha)$	$= @_t\,\Theta(\alpha)$	$@_u\alpha;\ t = \Theta(u)$
$\Theta(\alpha[u])$	$= \Theta(\alpha)[\Theta(u)]$	queries $\alpha[u]$

Table 1. Definition of substitution Θ based on query assignment (σ, τ)

3.2 Queries

We are now going to define the semantics of queries over streams.

Definition 9 (Query) *Let $S = (T, \upsilon)$ be a stream, $u \in T \cup \mathcal{U}$ and let α be a formula. Then $\alpha[u]$ denotes a* query (on S)*. We say a query is* ground*, if α is ground and $u \in T$, else* non-ground*.*

For the evaluation of a ground query $\alpha[t]$ we will use $M, S_M, t \Vdash \alpha$. To define the semantics of non-ground queries, we need the notions of assignments and substitution. A *variable assignment* σ is a mapping $\mathcal{V} \to \mathcal{C}$ from variables to constants. A *time variable assignment* τ is a mapping $\mathcal{U} \to \mathbb{N}$ from time variables to time points. The pair (σ, τ) is called a *query assignment*. Table 1 defines the *substitution* Θ based on query assignment (σ, τ), where $\alpha, \beta \in \mathcal{F}_k$.

Let $q = \alpha[u]$ be a query on $S = (T, \upsilon)$. We say a substitution Θ *grounds* q, if $\Theta(q)$ is ground, i.e., if Θ maps all variables and time variables occurring in q. If, in addition, $\tau(x) \in T$ for every time variable $x \in \mathcal{U}$ occurring in q, we say Θ is *compatible* with q.

Definition 10 (Answer) *The* answer $?q$ *to a query $q = \alpha[t]$ on S is defined as follows. If q is ground, then $?q = yes$ if $M, S_M, t \Vdash q$ holds, and $?q = no$ otherwise. If q is non-ground, then*

$$?q = \{(\sigma, \tau) \mid \Theta \textit{ is compatible with } q \textit{ and } ?\Theta(q) = yes\}.$$

That is, the answer to a non-ground query is the set of query substitutions such that the obtained ground queries hold.

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

$$q_1 = \boxplus_1(\Diamond tr(X, P) \wedge \Diamond bus(Y, P))[u]$$
$$q_2 = \boxplus_1\Diamond(tr(X, P) \wedge bus(Y, P))[u]$$

The query $q = \boxplus_1\Diamond(tr(a, p_1) \wedge bus(c, p_1))[t]$ is ground iff $t \in \mathbb{N}$ and $?q = yes$ iff $t \in [2, 7]$. We evaluate q_1 on structure M of Ex. 6:

$$M, S_M, t \Vdash \boxplus_1(\Diamond tr(a, p_1) \wedge \Diamond bus(c, p_1)) \text{ for all } t \in [2, 7]$$
$$M, S_M, t \Vdash \boxplus_1(\Diamond tr(d, p_2) \wedge \Diamond bus(e, p_2)) \text{ for all } t \in [11, 13]$$

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

$$?q_1 = \{(\{X \mapsto a, Y \mapsto c, P \mapsto p_1\}, \{u \mapsto t\}) \mid t \in [2, 7]\} \cup \{(\{X \mapsto d, Y \mapsto e, P \mapsto p_2\}, \{u \mapsto t\}) \mid t \in [11, 13]\}$$ ∎

Exact time reference. With the operator $@_t$ 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 $\alpha = tram(X, P) \wedge bus(Y, P)$. For each of the queries $@_U\alpha[13]$ and $\alpha[U]$, 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 $(\{X \mapsto a, Y \mapsto c, P \mapsto p_1\}, \{U \mapsto 2\})$. Note that omitting $@_U$ in the first query would give an empty answer, since the subformula α does not hold at time point 13. ∎

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

$$q_3 = \boxplus_1 \Box(tr(X, P) \rightarrow \boxplus_2 \Diamond bus(Y, P))[13].$$

For $\boxplus_1$, we can use the extension $\hat{w}^{\#n}_{\text{idx}}$ of the partition-based window $w^{\#n}_{\text{idx}}$ of Example 5. Applying $\hat{W}(1)$ on the stream $S = (T, \upsilon)$ in the previous examples yields $S' = (T', \upsilon')$, where $T' = [2, 13]$ and $\upsilon' = \{2 \mapsto \{tr(a, p_1)\}, 8 \mapsto \{tr(d, p_2)\}\}$. 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 urstream S_M. Thus, the second extended window $\hat{W}(2) = \hat{w}^{+3}$ is the extension of the time-based window w^{+3}, which looks 3 minutes into the future, due to the stream choice ch_1. Hence, $\hat{w}^{+3}$ 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 matching p_1 and p_2, respectively. Applying $\hat{W}(2)$ there yields the streams $S''_2 = (T''_2, \upsilon''_2)$ and $S''_8 = (T''_8, \upsilon''_8)$, where

$$T''_2 = [2, 5], \quad \upsilon''_2 = \{2 \mapsto \{tr(a, p_1), bus(c, p_1)\}\}, \text{ and}$$
$$T''_8 = [8, 11], \quad \upsilon''_8 = \{8 \mapsto \{tr(d, p_2)\}, 11 \mapsto \{bus(e, p_2)\}\}.$$

In both streams, we find a time point with an atom $bus(Y, p_j)$ with the same stop p_j as the tram. Thus, in both cases the subformula $\Diamond bus(Y, P)$ is satisfied and so the implication $tr(X, P) \rightarrow \boxplus_2 \Diamond bus(Y, P)$ holds at every point in time of the stream selected by $\boxplus_1$. Hence, the answer to the query is

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

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 $\Diamond$ and $\Box$ 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 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* 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 $\boxplus_i$ 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 $\Diamond x$ 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 (σ, τ). 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:

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

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

```
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.

```
SELECT * FROM tr  [ROWS 2],
              bus [RANGE UNBOUNDED]
WHERE tr.P = bus.P AND bus.TS - tr.TS <= 3
```

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_3) 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 $\Box$. With according rewriting, CQL eingines 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 recent 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_j), called *online progression* are used. While P and E_i capture accumulated knowledge, Q and F_j 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_j 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 $@_t$ 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 rules 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 $@_t p(x_1, \ldots, x_n)$ 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 (idealistic) 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 exsisting 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

[1] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic., 'Stream reasoning and complex event processing in ETALIS', *Semantic Web Journal*, (2012).

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom, 'Stream: The stanford stream data manager', in *SIGMOD Conference*, p. 665, (2003).

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom, 'The CQL continuous query language: semantic foundations and query execution', *VLDB J.*, **15**(2), 121–142, (2006).

[4] Shivnath Babu and Jennifer Widom, 'Continuous queries over data streams', *SIGMOD Record*, **3**(30), 109–120, (2001).

[5] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus, 'C-SPARQL: a continuous query language for rdf data streams', *Int. J. Semantic Computing*, **4**(1), 3–25, (2010).

[6] Patrick Blackburn, Maarten de Rijke, and Yde Venema, *Modal Logic*, Cambridge University Press, New York, NY, USA, 2001.

[7] Gerhard Brewka, 'Towards reactive multi-context systems', in *LPNMR*, pp. 1–10, (2013).

[8] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan, 'Modeling the execution semantics of stream processing engines with secret', *VLDB J.*, **22**(4), 421–446, (2013).

[9] Thang M. Do, Seng Wai Loke, and Fei Liu, 'Answer set programming for stream reasoning', in *AI*, pp. 104–109, (2011).

[10] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits, 'dlvhex: A prover for semantic-web reasoning under the answer-set semantics', in *Web Intelligence*, pp. 1073–1074, (2006).

[11] Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub, 'Stream reasoning with answer set programming', in *KR*, (2012).

[12] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub, 'Reactive answer set programming', in *LPNMR*, pp. 54–66, (2011).

[13] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele, 'Engineering an incremental asp solver', in *ICLP*, pp. 190–205, (2008).

[14] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth, 'A native and adaptive approach for unified processing of linked streams and linked data', in *ISWC (1)*, pp. 370–388, (2011).

[15] Danh Le Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink, 'Linked stream data processing engines: Facts and figures', in *ISWC - ET*, (2012).

[16] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel, 'It's a streaming world! reasoning upon rapidly changing information', *IEEE Intelligent Systems*, **24**, 83–89, (2009).

[17] Carlo Zaniolo, 'Logical foundations of continuous query languages for data streams', in *Datalog*, pp. 177–189, (2012).