Window Specification over Data Streams

Kostas Patroumpas and Timos Sellis

School of Electrical and Computer Engineering
National Technical University of Athens, Hellas
{kpatro,timos}@dbnet.ece.ntua.gr

Abstract. Several query languages have been proposed for managing data streams in modern monitoring applications. Continuous queries expressed in these languages usually employ windowing constructs in order to extract finite portions of the potentially unbounded stream. Explicitly or not, window specifications rely on ordering. Usually, timestamps are attached to all tuples flowing into the system as a means to provide ordered access to data items. Several window types have been implemented in stream prototype systems, but a precise definition of their semantics is still lacking. In this paper, we describe a formal framework for expressing windows in continuous queries over data streams. After classifying windows according to their basic characteristics, we give algebraic expressions for the most significant window types commonly appearing in applications. As an essential step towards a stream algebra, we then propose formal definitions for the windowed analogs of typical relational operators, such as join, union or aggregation, and we identify several properties useful to query optimization.

1 Introduction

Data streams have emerged as a modern paradigm for managing time-varying, volatile, unpredicted and possibly unbounded information in various monitoring applications. Typical examples include data generated from telecom calls, financial tickers, sensor readings over large areas or traffic measurements. Such information must be handled online as data items flow rapidly into the system from multiple sources. Over this dynamic data, the system must provide timely and incremental responses to multiple *continuous queries*, ideally keeping in pace with the data arrival rate.

Compared to one-time queries in conventional DBMS's, continuous queries differ substantially in their semantics. Since the size of the stream is potentially unbounded, the state of the data is not known in advance, so responses clearly depend on the set of stream tuples available during query evaluation. Several operations, such as aggregation or join between streams, may need special treatment, e.g., previously arrived tuples must be maintained, at the expense of a significant overhead. Obviously, since streaming data is usually retained in memory and not physically stored on disk, it is not practically feasible to "remember" the entire history of rapidly accumulating stream elements due to resource limitations. Besides, it is probably not worth maintaining stale tuples for long, as the significance of each isolated item is time-decaying for most realistic applications.

To overcome such difficulties, *windows* have been introduced in query formulation. Such constructs generally emphasize on the latest data by taking advantage of an ordering among tuples, usually established through timestamp values attached to every item. Intuitively, at any time instant, a window operator specifies a finite set of recent tuples from the unbounded stream; this finite portion of the stream will be subsequently used to evaluate the query and produce results corresponding to that time instant. As time advances, fresh items get included in the window at the expense of older tuples that stop taking part in computations (and perhaps may get discarded altogether). In general, windows evolve in a prescribed mode keeping up with the continuous arrival of data items. Certain variants have been suggested for effective stream processing, like *sliding*, *landmark* or *tuple-based* windows, whereas several types have been successfully implemented in prototype systems [1, 6, 8, 13]. However, most of these constructs are described in an abstract manner or by giving motivating examples, without paying particular attention to their subtle semantics.

The structure of a window clearly determines the snapshot of the dataset over which the query will be evaluated each time. But how can the extent of that window be defined in order to obtain a particular portion of the stream? Should this extent be allowed to change over time and in which specific pattern? Should window's contents be overlapping between any two successive snapshots? Finally, is it possible for windowing constructs to be intertwined with relational operators and under what semantic interpretation for their results?

In this paper, we provide a foundation for window specification over data streams with precise semantics. First, we sketch out a simple, yet quite robust model for querying data streams that is capable enough to capture their volatile nature. This model adheres to well-founded relational concepts and extends recent approaches in data stream management. Our contributions are as follows:

- We identify certain generic properties of windows that offer a sound basis for their taxonomy in several categories. Through these properties we can then create a mechanism for expressing typical window variants over streams.
- We introduce a parameterized *scope function* as a building block that enables effective specification of both the window's extent and its progression across time. Composite window types may then be defined by simply arranging their basic features according to the semantics of the query at hand.
- We provide formal definitions of well-known relational operators combined with windowing specifications that can be used to express queries on streams. We further investigate characteristic properties that may prove useful to query formulation and transformation.

The remainder of this paper is organized as follows: Section 2 presents a framework for data stream modeling and explains the semantics of continuous queries. In Section 3 a basic algebraic representation is introduced for windows according to their taxonomy into distinct types. Section 4 proposes a combination of windowing constructs with certain relational operators outlining several useful properties. Related work is briefly reviewed in Section 5. Finally, Section 6 offers conclusions and hints to future research directions.

2 A Framework for Querying Data Streams

2.1 Basic Notions on Data Streams

Items of a data stream are commonly represented as relational tuples [3], not excluding a semistructured form [16]. Henceforth we opt for a specification of stream items as relational tuples:

Definition 1 (Schema of tuples). *The tuple schema E of streaming items is represented as a set of elements $\langle e_1, e_2, \ldots, e_N \rangle$ of finite* arity *N. Each element e_i is termed* attribute *with name A_i and its values are drawn from a possibly infinite atomic* data type domain *D_i. Every tuple is an instance of the schema and it is described by its values at the respective attributes.*

A timestamp value is attached to every streaming tuple as a means of providing order among the data items that interminably flow into the system. It is often convenient to represent time as an ordered sequence of distinct moments (like clock ticks). Alternatively, simple sequence numbers may also serve as a means for ordering tuples, i.e., a unique serial number is attached to each tuple upon admission to the system. The following definition covers both interpretations:

Definition 2. Time Domain $\mathbb{T}$ *is regarded as an ordered, infinite set of discrete time instants $\tau \in \mathbb{T}$. A time* interval *$[\tau_1, \tau_2] \in \mathbb{T}$ consists of all distinct time instants $\tau \in \mathbb{T}$ for which $\tau_1 \leq \tau \leq \tau_2$.*

From the definition above, it follows that $\mathbb{T}$ may be considered similar to the domain of natural numbers $\mathbb{N}$. The extent of each interval is also a natural number, as it is simply the count of all distinct time instants occurring between its bounds. At each timestamp $\tau \in \mathbb{T}$, a possibly large, but always finite number of data elements of the stream arrive for processing [3]. Thus, multiset (bag) semantics apply and duplicates are allowed, signifying that zero, one or multiple identical tuples may arrive at any single instant:

Definition 3 (Data Stream). *A Data Stream S is a mapping $S : \mathbb{T} \rightarrow 2^R$ that at each instant $\tau \in \mathbb{T}$ returns a finite subset from the set R of tuples with common schema E. A supplementary attribute A_τ (not included in E) is designated as the* timestamp *of tuples and takes its ever-increasing values from $\mathbb{T}$.*

Note that other temporal indications (e.g., attached at data sources) are still allowed in the schema, although not explicitly considered as timestamps. In contrast, timestamp is a distinctive attribute attached to every stream element.

From a historical perspective, a data stream may be regarded as an ordered sequence of elements evolving in time, so its *current contents* are all tuples accumulated so far. On the other hand, an *instance* of the stream at any distinct time instant is a finite multiset of tuples with that specific timestamp value.

Definition 4. Current Stream Contents *$S(\tau_i)$ of a Data Stream S at time instant $\tau_i \in \mathbb{T}$ is the set $S(\tau_i) = \{s \in S : \ s.A_\tau \leq \tau_i\}$.*

Definition 5. Current Stream Instance *$S_I(\tau_i)$ of a Data Stream S at time instant $\tau_i \in \mathbb{T}$ is the set $S_I(\tau_i) = \{s \in S : \ s.A_\tau = \tau_i\}$.*

Timestamps serve as a unique time indication for the entire tuple and also as a common time reference for all incoming tuples. Each tuple maps to exactly one timestamp, but multiple tuples can have identical timestamp values. Timestamps cannot be assigned a `NULL` value. Hence, a total order of stream items may be defined by taking advantage of properties inherent in Time Domain:

Definition 6. Temporal Ordering *is defined as a many-to-one mapping $f_O : D_S \rightarrow \mathbb{T}$ from data type domain D_S of the tuples belonging to a data stream S to Time Domain $\mathbb{T}$, with the following timestamp properties:*
i) Existence: $\forall\ s \in S, \exists\ \tau \in \mathbb{T}$, *such that* $f_O(s) = \tau$.
ii) Monotonicity: $\forall\ s_1, s_2 \in S$, *if* $s_1.A_\tau \leq s_2.A_\tau$, *then* $f_O(s_1) \leq f_O(s_2)$.

Temporal ordering is crucial in stream processing because data items must be given for processing in accordance to their timestamps. As a general rule when evaluating a continuous query at time τ, all stream tuples with timestamps upto that particular τ must be available. Hence, no item should propagate for further execution if its timestamp value is less than the latest tuple produced by the system. Handling out-of-order tuples is beyond the scope of this paper.

2.2 Semantics of Continuous Queries

Intuitively, the results of a continuous query on a data stream may be considered as a union of the sets of tuples returned from successive query evaluations over the current stream contents at every distinct time instant. Similarly to [7, 19], we may formally define:

Definition 7 (Continuous Query over Stream). *Let Q a continuous query submitted at time instant $\tau_0 \in \mathbb{T}$ on data stream S. The results Q^c that would be obtained at $\tau_i \in \mathbb{T}$ are the union of the subsets $Q(S(\tau))$ of qualifying tuples produced from a series of one-time queries Q on successive stream contents $S(\tau)$:*

$$\forall\ \tau_i \in \mathbb{T},\ \tau_i \geq \tau_0,\ Q^c(S(\tau_i)) = \bigcup_{\tau_0 \leq \tau \leq \tau_i} Q(S(\tau))$$

The problem with this evaluation method is that it may not be practically feasible each time to compute query results by taking into account all stream contents due to the overwhelming bulk of data that keep accumulating continuously. Periodic evaluation is no better: if only intermediate stream contents are considered in each evaluation, it may happen that newer results may cancel tuples included in formerly given answers.

A conservative approach is to accept queries with *append-only results*, thus not allowing any deletions or modifications at answers already produced. This class of continuous queries is called *monotonic* [7]:

Definition 8 (Monotonic Continuous Query over Stream). *A continuous query Q applied over data stream S is characterized* monotonic *when*

$$\forall\ \tau_1, \tau_2 \in \mathbb{T},\ \tau_1 \leq \tau_2,\ \textit{if}\ S(\tau_1) \subseteq S(\tau_2),\ \textit{then}\ Q(S(\tau_1)) \subseteq Q(S(\tau_2)),$$

where $Q(S(\tau_i))$ denotes results for query Q that have been produced from qualifying tuples of stream contents $S(\tau_i)$ at time instant τ_i.

Obviously, the above definitions can be generalized for multiple input streams. It is important to note that monotonicity refers to query results and not to incoming stream items. As long as tuples may only be added to, but never discarded from results, incremental evaluation of queries involving projections or selections may be carried out as simple filters without particular complications. However, joins or set-theoretic operations may involve stream items that have arrived at previous time instants, so a state must be continuously maintained for them [20]. *Blocking operators*, like aggregation or sorting, cannot produce even a single tuple of their result before reading the entire input. *Stateful operators*, like join or intersection, are equally problematic: in order to execute a join, all tuples from both streams must be maintained, just in case a newly arriving data item matches an older tuple from the other stream.

In order to bound the increasing memory requirements of query operators, sliding windows are usually applied over the infinite streams and always return a finite portion of the most recent data items. However, continuous queries specifying sliding windows over streams are *non-monotonic*, since new results are produced but some older ones get expired due to window movement [10]. Regarding evaluation of sliding window queries, the interesting idea of *negative tuples* [11] has been suggested as a way to cancel previously returned, but no longer valid results. Certainly, this approach entails revision of typical query operators to make them capable to handle positive and negative tuples alike.

The exact role of relational tables in continuous queries is another concern. Whereas in Gigascope [13] there is no support for relations, other approaches allow static tables (e.g., AURORA [1]) or even arbitrary updates in time-varying relations (as in STREAM [3]). In the latter case, insertion and deletion tuples are used to represent the changing state of such a relation. In [10] it is suggested the notion of *non-retroactive relations*, whose updates affect only upcoming results but do not alter any previously given query answers. In this work, our focus is strictly on streams, setting aside for future research their interaction with relations.

We think that our proposition for window semantics presented in the next sections is flexible enough to be used under any of the aforementioned interpretations of continuous queries. Our main focus is on precise specification of windows, i.e., stream portions that may be regarded as temporary relations where typical operators may be applied under well-known relational semantics.

3 An Algebraic Representation for Windows

3.1 Window Semantics and Properties

In all data stream prototype systems, submission of continuous queries is always accompanied by –mostly sliding– window specifications on any stream involved in the query. A *window* is generally considered as a mechanism for adjusting

flexible bounds on the unbounded stream in order to fetch a finite, yet ever-changing set of tuples, which may be regarded as a temporary relation.

Definition 9 (Window over Data Stream). *Let W_E a window with conjunctive condition E applied at time instant $\tau_0 \in \mathbb{T}$ over the items of a data stream S, i.e., over its current contents $S(\tau_0)$. Then:*

$$\forall\ \tau_i \in \mathbb{T},\ \tau_i \geq \tau_0,\ W_E(S(\tau_i)) = \{s \in S(\tau_i) :\ E(s, \tau_i)\ holds\}$$

provided that $\mid W_E(S(\tau_i)) \mid \leq n$*, for any large, but always finite* $n \in \mathbb{N}$.

Therefore, each window is applied over the items of a single data stream S and at every τ_i returns a concrete finite set of tuples $W_E(S(\tau_i)) \subset S(\tau_i)$ which is called the *window state* at this time instant. When a continuous query involves multiple streams (e.g., joins), a separate window must be specified for each one, even if identical semantics are applied to all of them (i.e., similar expressions E).

Window specification is achieved by means of a *windowing attribute* [16] that helps in establishing order among stream items. We adhere to timestamps for ordering stream elements, so in the discussion below we designate timestamp attribute as the one used for obtaining tuples qualifying for condition E. Conjunctive condition E clearly depends on the windowing attribute and in our framework it takes the form of a *scope function*. This condition determines the exact structure of the window through its distinctive properties:

- *upper bound*: the timestamp of the most recent data item of the window, i.e., the greatest time indication or sequence number. Note that this is not necessarily the current timestamp, as the window may be delayed by an offset with regard to the most recent stream tuple.
- *lower bound*: the timestamp of the oldest data item within the window.
- *extent*: the "size" of the window, that may be expressed either as the number of tuples included in it or as the temporal interval spanning its contents.
- *mode of adjustment* as time advances. This crucial property determines whether and in what way a window changes state over time.

These properties are useful for classifying windows into distinct types according to the following criteria that prescribe their evolution with time:

Measurement Unit. Provided that one of its bounds is specified (e.g., the current time instant) a window can be described through its size, such that its contents may be obtained indirectly through their relative position to the known bound. Therefore, the *scope* of the window can be measured in:

- *logical units*, usually in timestamp values. Hence, a *time-based* window is derived from the time interval spanning its contents.
- *physical units*, implying the number of tuples falling within window's bounds. Typical variants include *count-based* (Fig. 1a) and *partitioned* windows.

Most often, the upper bound is known, as it can be derived easily either from the current timestamp value (under the logical interpretation) or the most recent tuple of the stream (physical interpretation) [6].

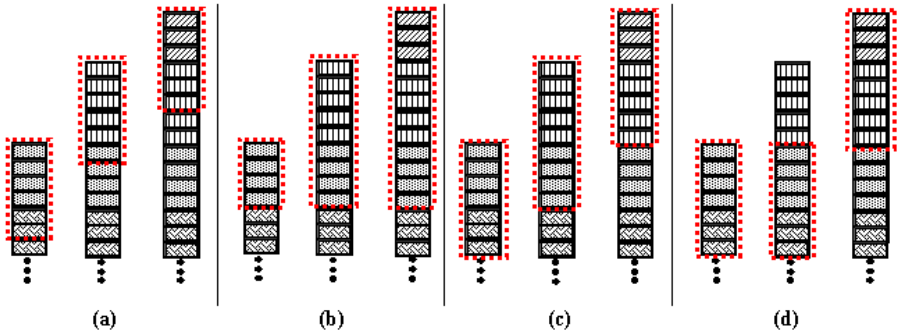

Fig. 1. Typical window variants illustrated for three consecutive states at time instants $\tau_k, \tau_k + 1, \tau_k + 2$. New data items are piling up on top of previously arrived ones. Boxes depicted with the same fill style represent tuples with identical timestamps. (a) *Count-based sliding* window of size $N = 6$. (b) *Landmark* window with lower bound fixed at τ_k. (c) *Sliding time-based* window of temporal extent $\omega = 2$ and progression step $\delta = 1$. (d) *Tumbling* window of temporal extent $\omega = 2$ and progression step $\delta = 2$.

Edge Shift. Alternatively, a window can be explicitly defined by its two bounds (or *edges*), which are commonly expressed as specific timestamp values for each state. Depending on whether edges can change over time, we distinguish between:

- *fixed-bound windows*, where at least one of the bounds remains anchored at a specific time instant. The other edge of the window is allowed to move freely. It is the upper bound that is usually shifted forward in pace with time progression, as occurs in *landmark windows* [8] (Fig. 1b).
- *variable-bound windows*, where both bounds change over time. For instance, in *sliding windows* both edges proceed in tandem at the same pace such that the window size (expressed either in time units or in tuple count) stays fixed.

Progression Step. Except for the case its bounds remain fixed, a window changes progressively its contents either due to the arrival of new streaming tuples or because of the advancement of time[1]. Therefore, transition between any two successive states of a window may be carried out at:

- *Unit step.* In that case, window's bounds advance smoothly one tuple-at-a-time or at every discrete time instant (assuming that a global clock exists). Overlaps should be expected between successive states of a window: when progression step is expressed in time units (*time-based sliding windows*), tuples with the oldest timestamp are discarded and those with the most recent

[1] In general, windows may be allowed to move not only forward in time, but also backwards. However, this approach is of little practical importance for applications that need to process data streams online. In this paper, we assume that windows move along the direction of increase in timestamp values.

timestamp get inserted; in general, the number of expired tuples is not necessarily equal to those appended (Fig. 1c). As for *count-based sliding windows*, each incoming tuple throws away the oldest one. In both cases, window's contents get modified only across its bounds, retaining all tuples in between.

- *Hops* that span multiple time instants or a specific number of tuples. Depending on whether this hop size is smaller or larger than the window size, overlapping or non-overlapping window extents may be created, respectively. For example, non-overlapping *tumbling windows* [1, 13] are used to get different portions of the stream, so that no data item takes part in calculations twice (Fig. 1d).

Many window variants can be specified on the basis of the aforementioned classification criteria, depending on the semantics of the respective continuous queries. For instance, a sliding time-based window (Fig. 1c) may be needed so that the time interval covered by its contents remains fixed, although the number of tuples within the window might be varying over time. Besides, conjunctive condition E that identifies qualifying tuples may be extended with additional filtering predicates on other attributes apart from the windowing one, in a way that *value-based windows* may be expressed [6]. However, the main motivation behind windowing constructs is their combination with typical relational operators (join, aggregation, etc.), so that their windowed analogs can be specified in continuous queries over data streams.

In the next subsections, we attempt a rigorous algebraic description of the principal window types that have been proposed in the context of data streams, assuming that timestamps are used as windowing attributes. We adopt from [6] the basic discrimination in *physical* and *logical windows*, according to the unit in which window contents are determined.

3.2 Physical Window Types

Since these windowing constructs are determined by a predefined number of tuples, they are sliding by default, so naturally, their extent spans the most recent stream elements ("backward"). We are not aware of any practical application that might ask for the N data items that will be arriving after the k-th element ("forward"), because it cannot be known in advance whether or even when this specific tuple will be observed in the flow of the unbounded stream. In the following, window states are determined only at single-tuple units, as it seems unlikely to specify a slide parameter of multiple tuples in most cases.

Count-based Windows. At every time instant $\tau \in \mathbb{T}$ a typical count-based window covers the most recent N tuples of stream S:

$$W_n(S, \tau, N) = \{s \in S(\tau) : \exists\ \tau_1 \in \mathbb{T}\ (\tau_1 \leq \tau \wedge\ \mid \{s \in S(\tau) :\ \tau_1 \leq s.A_\tau \leq \tau\} \mid \leq N) \\ \wedge \forall\ \tau_2 \in \mathbb{T}\ (\tau_2 < \tau_1 \wedge\ \mid \{s \in S(\tau) :\ \tau_2 \leq s.A_\tau \leq \tau\} \mid > N)\}$$

The above formula implies the method utilized to identify qualifying tuples: intuitively, starting from the current time instant τ and going steadily backwards

in time, tuples are being obtained until their total count exceeds threshold N (cf. Fig. 1a for a graphical representation of such a window with size $N = 6$ tuples). Nevertheless, subtle issues may arise with this policy. When the contents of count-based windows are derived through their sequence numbers [3], it must be clear how many times possible duplicates are counted and how ties are broken for the N-th element. A similar case, but concerning timestamped tuples, arises in our definition above: ties may still occur when only k elements need be chosen out of a batch of $m > k$ tuples corresponding to the lower bound of the window, in order to reach the predefined total count N. As a convenient workaround to resolve both subtleties, tuples may be selected in a non-deterministic fashion, as suggested for `ROW`-based windows in CQL [3].

Partitioned Windows. The semantics of this window type are applied to the streaming tuples by first partitioning them according to a subset $L = \{A_1, A_2, \ldots, A_k\}$ of *grouping attributes*, as in extended relational algebra. Therefore, several *substreams* are derived, each one corresponding to an existing combination of values $\langle a_1, a_2, \ldots, a_k \rangle$ on the grouping attributes. From each resulting partition the most recent N elements are taken and the union of these subsets provides the final set of window tuples. Note that the windowing attribute (timestamp) is not allowed to participate in the list of grouping attributes. Formally, this operation may be defined as follows:

$$\begin{aligned} W_p(S, \tau, L, N) = \{s \in S(\tau) : \forall A_k \in L,\ s.A_k = a_k\ \wedge\ \ a_k \in D_k\ \wedge \\ \wedge\ \exists\ \tau_1 \in \mathbb{T}\ (\tau_1 \leq \tau\ \wedge\ \mid \{s \in S(\tau) :\ s.A_k = a_k\ \wedge\ \tau_1 \leq s.A_\tau \leq \tau\} \mid \leq N) \wedge \\ \wedge\ \forall\ \tau_2 \in \mathbb{T}\ (\tau_2 < \tau_1\ \wedge\ \mid \{s \in S(\tau) :\ s.A_k = a_k\ \wedge\ \tau_2 \leq s.A_\tau \leq \tau\} \mid > N)\} \end{aligned}$$

In contrast to usual relational semantics, aggregate functions (like `SUM`, `AVG`, etc.) are not applied to the partitions formed after grouping stream elements. Instead, a subset of N tuples is obtained from each partition and not just a single value expressing their total count. Observe that *count-based windows* may be regarded as a special case of partitioned windows where all tuples of the stream get assigned to a single partition with no grouping attributes specified.

3.3 Logical Window Types

In logical windows, the timestamp values of streaming tuples are checked for inclusion within a prespecified temporal interval. We conveniently express this requirement by means of a *scope function* that may be defined for each window type as a mapping from Time Domain $\mathbb{T}$ to the domain of possible time intervals:

$$scope :\ \mathbb{T} \rightarrow \{[\tau_1, \tau_2] : \tau_1, \tau_2 \in \mathbb{T},\ \tau_1 \leq \tau_2\}$$

Essentially, at every time instant the scope function returns the window bounds (and *not* its actual contents), taking as parameters the properties of the respective window type (extent, progression step, etc.).

Due to lack of space, in the following we present the most representative variants of logical windows that have been implemented for several stream prototypes, although the expressiveness of this approach has a broader applicability.

Landmark Windows maintain one of their bounds fixed at a specific time instant, letting the other follow the evolution of time. We distinguish two cases:

Lower-bounded landmark window. The lower bound (i.e., the starting time τ_l) of the window is permanent, whereas the upper bound proceeds with time. If this construct is applied at time τ_0, then at any subsequent time $\tau \geq \tau_0 \in \mathbb{T}$ the scope function takes the form:

$$scope_l(\tau) = \begin{cases} \emptyset & \text{if } \tau_0 \leq \tau < \tau_l \\ [\tau_l,\ \tau] & \text{if } \tau_0 \leq \tau_l \leq \tau \end{cases}$$

Therefore, streaming tuples of S with timestamps that qualify for the scope of this landmark window are returned as its state at every time instant:

$$W_l(S, \tau, \tau_0, \tau_l) = \{s \in S(\tau) :\ s.A_\tau \in scope_l(\tau)\}$$

Note that this window type will keep appending new tuples indefinitely, unless either the query is explicitly revoked (and hence the window is cancelled) or the stream is exhausted and no tuples enter into the system anymore.

Upper-bounded landmark window. Here it is the upper edge that has been fixed to a future time instant, which might not yet have occurred, but it will eventually occur due to time monotonicity. Assuming that such a window is applied at time τ_0, the scope function is as follows:

$$scope_u(\tau) = \begin{cases} [\tau_0,\ \tau] & \text{if } \tau_0 \leq \tau < \tau_u \\ [\tau_0,\ \tau_u] & \text{if } \tau_0 \leq \tau_u \leq \tau \end{cases}$$

and the upper-bounded landmark window is defined accordingly:

$$W_u(S, \tau, \tau_0, \tau_u) = \{s \in S(\tau) :\ s.A_\tau \in scope_u(\tau)\}$$

Of course, there is no point in specifying upper bounds in the past ($\tau_u < \tau_0$). Intuitively, as long as the upper bound has not been reached yet, the scope of window keeps expanding. After time instant $\tau = \tau_u$, the scope will no longer change, so the window will "close" and its bounds will be fixed. For append-only streams, this means that the contents of the upper-bounded window will thereafter be "frozen", like a materialized snapshot of a particular stream portion.

Fixed-band Windows. Combining the aforementioned landmark window variants, a *band window* function with fixed upper and lower bounds is constructed:

$$scope_b(\tau) = \begin{cases} \emptyset & \text{if } \tau < \tau_l \\ [\tau_l,\ \tau] & \text{if } \tau_l \leq \tau \leq \tau_u \\ [\tau_l,\ \tau_u] & \text{if } \tau_u < \tau \end{cases}$$

and $$W_b(S, \tau, \tau_l, \tau_u) = \{s \in S(\tau) :\ \tau_l \leq \tau_u\ \wedge\ s.A_\tau \in scope_b(\tau)\}$$

Note that the state is not related to the moment τ_0 this window is initially applied, hence even arbitrary time intervals ("bands") in the past may be expressed. From a semantics point of view, as soon as the current stream timestamp τ exceeds the upper bound (third branch), window's contents will remain unchanged, assuming that they can be maintained in memory indefinitely.

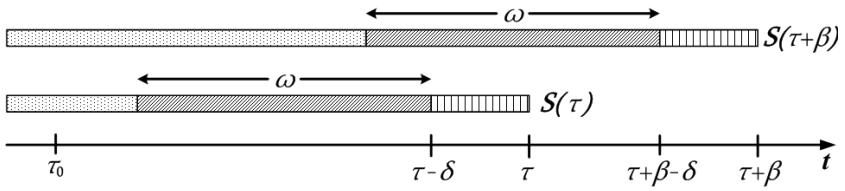

Fig. 2. Two successive states of a *sliding window* at time instants τ and $\tau + \beta$. This window obtains tuples delayed by δ time units with regard to current time (i.e., lagged elements). Since $\beta < \omega$, window states (ω) may have overlapping tuples. In case $\beta \geq \omega$, there is no longer smooth state transition, thus a *tumbling window* is actually applied.

Time-based Sliding Windows. This is probably the most common class of windows over data streams, defined by means of time units (recall that physical windows also slide as new tuples arrive). Let $\tau_0 \in \mathbb{T}$ be the time instant that a continuous query is initially submitted specifying a sliding window W_s. Let ω denote the invariable temporal extent of this window, β its progression step and suppose that the upper bound of the window has a delay (or *lag*) δ with regard to the current time instant τ. Then the scope of this sliding window may be defined as a function of time:

$$scope_s(\tau) = \begin{cases} \emptyset & \text{if } \tau_0 \leq \tau < \tau_0 + \delta \\ [\tau_0,\ \tau - \delta] & \text{if } \tau_0 \leq \tau - \delta < \tau_0 + \omega\ \wedge\ \texttt{mod}((\tau - \tau_0), \beta) = 0 \\ [\tau - \delta - \omega + 1,\ \tau - \delta] & \text{if } \tau \geq \tau_0 + \delta + \omega\ \wedge\ \texttt{mod}((\tau - \tau_0), \beta) = 0 \\ scope_s(\tau - 1) & \text{if } \texttt{mod}((\tau - \tau_0), \beta) \neq 0 \end{cases}$$

In the most common case, the upper bound of the sliding window coincides with the current timestamp of the stream (i.e., $\delta = 0$), so the previous scope function may be simplified as follows:

$$scope_s(\tau) = \begin{cases} [\tau_0,\ \tau] & \text{if } \tau_0 \leq \tau < \tau_0 + \omega\ \wedge\ \texttt{mod}((\tau - \tau_0), \beta) = 0 \\ [\tau - \omega + 1,\ \tau] & \text{if } \tau \geq \tau_0 + \omega\ \wedge\ \texttt{mod}((\tau - \tau_0), \beta) = 0 \\ scope_s(\tau - 1) & \text{if } \texttt{mod}((\tau - \tau_0), \beta) \neq 0 \end{cases}$$

Note that $\tau_0, \tau \in \mathbb{T}$ are expressed in timestamp values, whereas parameters ω, δ, β are actually sizes of time intervals (hence $\omega, \delta, \beta > 0$). For the sake of clarity, all parameters may be considered as natural numbers according to the definition of the Time Domain $\mathbb{T}$, so the scope function is evaluated at discrete time instants of $\mathbb{T}$. For every time instant $\tau \in \mathbb{T}$, the qualifying tuples are included in the window state:

$$W_s(S, \tau, \tau_0, \omega, \beta, \delta) = \{s \in S(\tau) :\ s.A_\tau \in scope_s(\tau)\}$$

In the most general case where $\beta < \omega$, overlaps are observed between the extents of any two successive states of a sliding window, thus a subset of their contents remains intact across states (common tuples in both ω in Fig. 2). In the meantime of any two successive evaluations that are β units apart, no change occurs

to the qualifying tuples. That is exactly the meaning of the recursive expression at the last branch of the function, which provides a warranty that window's bounds change discontinuously at time instants that depend strictly on the pattern stipulated by the progression step β. The definition allows for the existence of "half-filled" windows with extent less than ω at early evaluation stages, so the window may be considered as being gradually filled with tuples. As soon as the extent reaches its capacity, the window starts exchanging some older tuples with newly arriving ones. Since time evolution implies an analogous change of time intervals derived from scope, this function is by definition *monotonic*. Function scope holds even for time instants in the future, thus covering all forthcoming stream elements, no matter when they will actually arrive for processing.

Progression step β is usually set equal to the granularity of time (e.g., seconds), so that the window slides smoothly in pace with the advancement of time. In that case, the recursive branch in the above definition for the scope function is redundant, as window's contents are modified at every time instant.

Finally, by setting $\tau = \texttt{NOW}$, $\omega = 1$, $\delta = 0$ and $\beta = 1$ in the definition of the scope function, it is very easy to express an important class of sliding windows that obtain the current instance $S_I(\tau)$ of the stream, i.e., all its tuples with the current timestamp value (in [3] the shortcut $S\,[\texttt{NOW}]$ is used for this purpose).

Time-based Tumbling Windows. The scope function defined for sliding windows is generic enough to express windows with arbitrary progression step (even $\beta \geq \omega$). Intuitively, *tumbling windows* accept streaming tuples in "batches" that span a fixed time interval. This is particularly useful when aggregates must be computed over successive, yet non-overlapping portions of the stream, in a way that no tuple takes part in computations more than once [13]. Usually, a new window state is created as soon as the previous one has ceased to exist: the lower bound of the current state and the upper bound of its preceding one are consecutive time instants. This variant can be derived by simply setting $\beta = \omega$ at the scope function $scope_s$ of a sliding window, assuming a standard extent ω is used. At each evaluation, disjoint stream portions of equal extent are returned and thus window contents are obtained in a discontinuous fashion:

$$W_t(S, \tau, \tau_0, \omega, \omega, \delta) = \{s \in S(\tau) :\ s.A_\tau \in scope_s(\tau)\}$$

Alternatively, for several applications (e.g., traffic monitoring), different window sizes might be needed (e.g., for peak or night hours, weekends etc.). In that case, function $scope_s$ is still valid by replacing fixed extent ω with a time-varying $\omega(\tau)$. In [1] tumbling windows may be accompanied with user-defined predicates so as to determine the end of temporary states, but this approach is mainly geared towards implementation efficiency rather than query semantics.

3.4 Monotonicity of Window Types

From the discussion above, it is apparent that monotonicity varies according to the characteristics of window types, since edge shift and progression step

clearly determine containment and expiration of timestamped tuples with regard to the window specified. In [10] a characterization on monotonicity has been introduced for query operators with respect to sliding windows. More specifically, sliding windows are described as *weakest non-monotonic*: although new tuples are appended to a window state pushing older ones out of it, order is always preserved, since tuples are included into and excluded from a sliding window (either time- or count-based) in a FIFO fashion.

Here, we will briefly comment upon monotonicity of the remaining window variants. In particular, partitioned windows are *weak non-monotonic*. Although the contents of each of its constituent substreams change in FIFO order, some partitions may be modified more often than others. In fact, depending on the pattern of incoming tuples, some combinations of values on the grouping attributes may be observed more frequently. As a result, the expiration order of tuples does not generally coincide to their insertion order into the window.

However, lower- and upper-bounded landmark windows are *monotonic*. In either case, no tuple is ever removed from window state. Therefore, at any time instant the window state subsumes all previous ones. Accordingly, fixed-band windows are also monotonic, as their bounds remain intact over time.

The case appears more intricate for tumbling windows. At a first glance, each window state has no overlapping tuples with its predecessor, so this type is clearly non-monotonic. However, every state ceases to exist in its entirety as soon as the new one is initiated, so it may be assumed that each participating tuple is removed from that window at the same order it was inserted, emulating some kind of deferred elimination. Therefore, tumbling windows may be considered as *weakest non-monotonic*, exactly like their sliding counterparts.

4 Windowed Queries over Data Streams

As already pointed out, the main motivation behind the introduction of windows is the necessity to unblock query operators in stream processing. In fact, the combination of windows with relational operators creates their windowed analogs that accept streams of timestamped tuples as input and generate temporary relations as answers. If resulting tuples must be reassembled as a stream for further processing, a converse *Streamline* operator (like `ISTREAM`, `DSTREAM`, `RSTREAM` proposed in [3]) is needed to progressively make up the derived stream. However, this transformation does not affect window semantics and it will not be further examined here. Derived items are always given suitable time indications, whose value is operation dependent. Next, we describe these windowed operators and briefly present some of their properties, but a meticulous investigation regarding the minimal set of operators for a stream algebra is left for future work.

4.1 Windowed Operators

Since projection and selection are neither blocking nor stateful operators, application of windows is not strictly necessary, because both operators act like

filters over each streaming tuple. However, in many circumstances query semantics either include window specification (e.g., maintain all sensor readings over the past hour) or even impose a suitable one to facilitate query execution [3].

Windowed Projection. We define *windowed projection* π_L^W as an operator that applies a window[2] W over the contents of a stream S and returns all qualifying tuples retaining a restricted set of chosen attributes $L = \{A_1, A_2, \ldots, A_k\}$:

$$\pi_L^W(S(\tau)) = \pi_L\ (W(S(\tau))) = \{\langle s.A_1, s.A_2, \ldots, s.A_k, s.A_\tau\rangle : s \in W(S(\tau))\}$$

This "vertical" operator treats the contents of each window state as a typical relation and it simply projects out any unnecessary attributes. Note that the timestamp values from attribute A_τ of input tuples are attached to the resulting ones as well.

This operation can be further extended to *generalized projection*, in which expressions involving attributes, constants or arithmetic operators may be computed by considering each tuple of S in turn. Occasionally, a *renaming* operator may be utilized to control the names of composite expressions that appear as attributes in derived streams, in the same sense as in relational algebra [2].

Windowed Selection. Assuming that a condition F will be applied to each state of a window W over stream S, the selection operator can be defined as

$$\sigma_F^W(S(\tau)) = \sigma_F\ (W(S(\tau))) = \{s \in W(S(\tau)) : F(s)\ holds\}$$

Condition F may be an *atomic* one, that is, either $s.A_i = a_i$ or $s.A_i = s.A_j$. In the former case, the value of an attribute A_i is checked for equality to an atomic value $a_i \in D_i$ from its data type domain. In the latter case, A_i, A_j can be any two distinct attributes (apart from the timestamp) in the schema of stream tuples. Further, a *generalized* condition F may be defined with comparison operators $\theta \in \{=, \neq, <, \leq, >, \geq\}$ or as a conjunction of atomic selections, exactly as in relational algebra [2], since each predicate is applied over the temporary relation derived from its respective window state. Note that the schema of each tuple is left intact by this "horizontal" operator, hence the original timestamp value is retained in attribute $s.A_\tau$ for each item s at the output.

Windowed Duplicate Elimination. This operator applies a window W over the contents of a stream S and returns the most recent appearance of each tuple, eliminating any other identical tuple within the current extent of W:

$$\delta^W(S(\tau)) = \delta\ (W(S(\tau))) = \\ \{s \in W(S(\tau)) : \nexists\ s' \in W(S(\tau)), \forall\ A_i \in E, s'.A_i = s.A_i \wedge s'.A_\tau \geq s.A_\tau\}$$

Note that a more conservative strategy can be adopted, which maintains each distinct tuple as long as its timestamp falls within window's extent. Only when this tuple expires, is it replaced by a more recent identical tuple [10].

[2] For clarity, we henceforth eliminate conjunctive condition E in window notation.

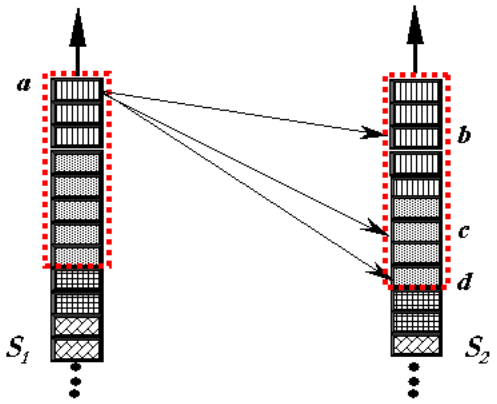

Fig. 3. Join operation between two streams S_1 and S_2 with different windows specified over each one. Each incoming tuple from either stream is tested for possible match with every tuple in the window applied to the other stream such that potential matches (pointed to with arrows) are returned.

Windowed Join. This symmetric binary operator may be applied between two streams (and easily generalized for multi-way joins), but there is no restriction that windows of the same type or the same scope must be specified over each stream[3]. At each time instant $\tau \in \mathbb{T}$, the *windowed join* between two streams returns the concatenation of pairs of matching tuples taken from either window state. In particular:

$$S_1(\tau) \overset{\bowtie}{}_{W} S_2(\tau) = W_1(S_1(\tau)) \bowtie W_2(S_2(\tau)) = \{\langle s_1, s_2, \tau_m \rangle : \\ s_1 \in W_1(S_1(\tau)),\ s_2 \in W_2(S_2(\tau)) \ \wedge\ J(s_1, s_2) \ \wedge\ \tau_m = \mathtt{min}(s_1.A_\tau, s_2.A_\tau)\}$$

As illustrated in Fig. 3, each newly arriving tuple within window W_1 of stream S_1 is checked for possible matches against the current state of window W_2 of stream S_2, and vice versa. Matching is performed according to the join condition J involving attributes from both streams (e.g., $S_1.A_i = S_2.A_j$). If matching tuples are found, the resulting joined element must be assigned a new timestamp value. Several policies have been suggested: in [10] the minimum of the two original timestamp values is given to the new tuple, with the natural interpretation that the concatenated tuple should expire from window as soon as one of the original tuples expire. Although this rule is acceptable from a semantics point of view (hence we adopt it in the definition above), it can lead to disorder among the joined tuples and to complications on further processing. Alternatively [4], the most recent from the pair of timestamp values attached to its constituent tuples can be chosen, as a means to preserve ordering in the derived stream. It has also been suggested that each resulting tuple may be

[3] In this paper we do not examine interaction of streams with static relational tables (e.g., in joins), as we consider that windows are applied solely over streams.

assigned to the time instant it was produced from the join operator [6], but this approach might be troublesome for successive joins in complex execution plans. In all cases, the chosen timestamp value substitutes existing ones at the concatenated tuple, so that only one timestamp attribute is retained.

Windowed Aggregation. Similarly to the respective operator in extended relational algebra, at first a grouping of window's tuples takes place according to their values for those attributes specified in grouping list $L = \{A_i, A_j, \ldots, A_n\}$. Next, for each combination of values $\langle a_i, a_j, \ldots, a_n \rangle$, an aggregation function f (like `COUNT`, `SUM`, `MIN`, `MAX` or `AVG`) is applied. If no attributes are specified, then all tuples in the window are regarded as belonging to a single group. Formally:

$$\gamma_L^{f^W}(S,\tau) = \gamma_L^f\ (W(S(\tau))) = \{\ \langle a_i, a_j, \ldots, a_n, f(a_i, a_j, \ldots, a_n), \tau_m \rangle :\ \tau_m = \tau\ \wedge \\ \forall\ A_k \in L, a_k \in D_k, s \in W(S(\tau))\ \wedge\ a_k = s.A_k\}$$

Note that the current time instant (assuming a global system clock exists) is attached to the resulting tuple as its timestamp. Alternatively, the most recent timestamp among all window state elements may be used, so that ordering can be achieved for the operator's output. More adequately, the $\mathtt{min}(s.A_\tau)$ among all current tuples $s \in W(S(\tau))$ participating in a group may be assigned as the timestamp τ_m for this group, but that may cause disorder to the derived stream.

Windowed Set-theoretic Operations. As it will become obvious from the following algebraic expressions, set-theoretic operations on data streams adhere to bag semantics as in relational algebra [9]. Therefore, we extent tuple schema with a positive integer value k that counts the number of duplicate stream elements within each window state. In the following, we denote as $\langle s, k \rangle$ a bag (multiset) where tuple s appears k times. Operations like *Windowed Union*, *Windowed Intersection*, and *Windowed Difference* are applied at each instant $\tau \in \mathbb{T}$ over the respective window states (even of diverse specifications), provided that both streams involved in these operations must have identical schemata:

$$S_1(\tau)\ \underset{W}{\bigcup}\ S_2(\tau) = W_1(S_1(\tau))\ \bigcup\ W_2(S_2(\tau)) = \{\langle s, \tau, k \rangle : \exists\ k_1, k_2 \in \mathbb{N}, \\ (\ \langle s, k_1 \rangle \in W_1(S_1(\tau))\ \vee\ \langle s, k_2 \rangle \in W_2(S_2(\tau))\ )\ \wedge\ k = k_1 + k_2\ \wedge\ k \neq 0\}$$

$$S_1(\tau)\ \underset{W}{\bigcap}\ S_2(\tau) = W_1(S_1(\tau))\ \bigcap\ W_2(S_2(\tau)) = \{\langle s, \tau, k \rangle : \exists\ k_1, k_2 \in \mathbb{N}, \\ \langle s, k_1 \rangle \in W_1(S_1(\tau))\ \wedge\ \langle s, k_2 \rangle \in W_2(S_2(\tau))\ \wedge\ k = \mathsf{min}(k_1, k_2)\ \wedge\ k \neq 0\}$$

$$S_1(\tau)\ \underline{^W}\ S_2(\tau) = W_1(S_1(\tau)) - W_2(S_2(\tau)) = \{\langle s, \tau, k \rangle : \exists\ k_1, k_2 \in \mathbb{N}, \\ \langle s, k_1 \rangle \in W_1(S_1(\tau))\ \wedge\ \langle s, k_2 \rangle \in W_2(S_2(\tau))\ \wedge\ k = \mathsf{max}(0, k_1 - k_2)\ \wedge\ k \neq 0\}$$

Observe that these operators essentially treat all copies of a tuple as being distinct to each other, so they simply manipulate their number of occurrences.

Also note that the timestamp attached to resulting tuples is the time instant of their evaluation, as existing timestamps are not taken into account when checking for duplicates. This might be reasonable since window contents are treated as transitory multisets. Alternatively, the k most recent (or the k older) tuples of each multiset may be returned each time, with the drawback of unsynchronized results produced from successive window states. Generalization of windowed union and intersection for more than two stream inputs is trivial.

4.2 Properties of Windowed Operators

Due to lack of space, we present just some indicative properties of the windowed operators defined previously, so as to emphasize their usefulness in query rewriting. First, it is obvious that *projection* is commutative with both logical and physical windows, that is, $\pi_L\ (W_E(S(\tau))) = W_E(\pi_L(S(\tau)))$.

Selection commutes with logical time-based windows only, i.e., $\sigma_F\ (W_E(S(\tau)))$ $= W_E(\sigma_F(S(\tau)))$, but not physical ones [3]. Evidently, the state of a count-based window may contain different items if selection has formerly been applied to the stream. Similarly, *duplicate elimination* also commutes solely with logical windows, i.e., $\delta\ (W_E(S(\tau))) = W_E(\delta(S(\tau)))$.

On the other hand, *stateful operators* like joins, intersections or aggregates, generally do not commute with any type of windows. Still, windowed analogs of binary operators have some interesting properties:

Rewriting Rules for Windowed Joins.

i) *Commutative:* $S_1(\tau) \overset{\bowtie}{}_W S_2(\tau) = S_2(\tau) \overset{\bowtie}{}_W S_1(\tau)$

ii) *Associative:* $(S_1(\tau) \overset{\bowtie}{}_W S_2(\tau)) \overset{\bowtie}{}_W S_3(\tau) = S_1(\tau) \overset{\bowtie}{}_W (S_2(\tau) \overset{\bowtie}{}_W S_3(\tau))$

iii) *Distributive over selection:* $\sigma_F(S_1(\tau) \overset{\bowtie}{}_W S_2(\tau)) = \sigma_F(S_1(\tau)) \overset{\bowtie}{}_W \sigma_F(S_2(\tau))$
Similarly to selections, this property holds for logical windows only.

iv) *Distributive over projection:* $\pi_L(S_1(\tau) \overset{\bowtie}{}_W S_2(\tau)) = \pi_L(\pi_{L_1}(S_1(\tau)) \overset{\bowtie}{}_W \pi_{L_2}(S_2(\tau)))$
Attribute lists L_1 and L_2 used for the separate projections over each stream must include attributes in list L and attributes involved in join conditions.

As for *Windowed Union* and *Windowed Intersection*, commutativity and associativity hold, but not distribution over selection. Note that union is not a blocking operator for streams, since its result may be produced in an incremental fashion by simply merging the current instances $S_I(\tau)$ of incoming streams.

5 Related Work

Stream processing has become a very fertile topic for database researchers over the past few years, but here we review issues mostly relative to continuous queries and window semantics. The first notion of continuous queries over append-only databases appeared in Tapestry [19] as a means to provide timely responses by

utilizing periodic query execution and identifying several rewriting rules for incremental evaluation. In [7], this approach was extended such that continuous semantics could deal with more involved cases, i.e., when deletions or modifications are allowed in the database, and not just insertions. Previous work in sequence databases [18] has provided useful semantics and a declarative language for managing ordered relations, but it is not too expressive for continuous queries over infinite streams. Besides, issues such as temporal modeling, ordering and indexing have been extensively studied in the context of temporal databases [12]. In [14] a temporal foundation for a stream algebra is attempted, which makes a distinction between logical and physical operator levels. Transformation rules are provided between a logical level that refers to query specification and a physical level that covers implementation issues. In terms of windows, only sliding and fixed variants are supported.

There have been several proposals for a query language over data streams. The declarative Continuous Query Language (CQL), which is being developed for the STREAM prototype [3], supports management of both dynamic streams and updatable relations and introduced mappings between them. CQL adopts semantics for queries and windows that are closest in spirit to ours. A more detailed approach on stream and query semantics is provided in [5], but windowing issues are covered briefly for tuple-based and time-based sliding windows only. StreaQuel is a SQL-like query language that is being developed for the TelegraphCQ project [8], although a subset of its functionality has been implemented so far. At present, continuous queries in TelegraphCQ may only be specified through time-based sliding windows, but there are plans to support more window variants (landmark, fixed, band). Several window types have also been implemented in Aurora [1], a prototype system that assumes a work-flow paradigm for data streams, instead of a relational one. Apart from windowed versions of join and sort operations, the main focus is on computing aggregates through sliding and tumbling windows. Gigascope [13] is a system for managing network flow in large data communication networks, where all stream tuples include an ordering attribute, exactly as in sequence databases. No windows are explicitly defined in its stream-only language GSQL, but their semantics can be indirectly expressed into constraints by analyzing the timestamps of input streams and query properties.

Most recently, window semantics are briefly touched in [16] in a more general setting for data stream representation, whereas a more detailed examination of window aggregates is proposed by the same authors in [15]. In brief, they attach explicit identifiers to all window states created during processing and they maintain the window states where each tuple actually participates in. This is performed by means of a special function that is the inverse of the one returning the extent. In essence, an additional attribute is attached to the grouping list used for aggregation; hence, windowed aggregation reduces to a simple relational one. One of their goals is to deal with disorder in incoming stream elements, hence they accept as windowing attribute any one with a totally ordered domain, i.e., not only timestamps or sequence numbers. Further, their view focuses mostly on

aggregates, and it covers only sliding, landmark and partitioned windows. The authors in [10] distinguish time-evolving streams from relational tables in order to derive update patterns for continuous queries. These patterns are classified according to monotonicity, in order to develop suitable physical query plans for processing and data structures for state maintenance. However, that insightful framework is limited to time-based sliding windows only, with no formal foundations. Although we set out with a similar overall approach on window semantics, we differ substantially in the development of window formalization. We avoid to assign identifiers to transient window states, since we think that it relates mostly to optimization issues rather than query semantics. We do not focus strictly on window interaction with aggregates, but we tackle all main relational operations. Further, we provide a detailed description of windows' properties and rich semantics for the most typical variants.

6 Conclusions and Future Work

In this paper we developed a foundation with clear semantics for specifying windows over data streams. To the best of our knowledge, our approach is the first to determine several common properties for windows, presenting a sound taxonomy of the most significant variants proposed in the literature. In order to overcome subtle intricacies, we introduced a generic scope function as a building block for effective specification of both the window's extent and its progression across time. More composite window types can then be defined by simply arranging their basic characteristics in consistency with query semantics. Further, our algebraic formulation for the windowed analogs of principal relational operators is of particular importance as a mechanism for expressing queries on streams and checking for syntactic equivalences. Overall, we believe that this approach is an essential step towards creating an algebra and a query language for managing data streams. Of course, several demanding topics remain open for future research, such as inclusion of relations or completeness of stream operators.

We are currently implementing (in C++) several operators to verify feasibility of our semantic foundations. We have begun developing a simplified stream processing engine for submitting continuous queries, which, of course, is far from a full-fledged DSMS. As of the time of writing this paper, all window variants have been successfully constructed, as well as windowed implementations for selection, projection and join in order to support typical SPJ continuous queries. Encouraged by these positive indications, we are in the process of gradually incorporating more operations, particularly aggregation and duplicate elimination.

We also believe that this framework is a promising area for research concerning *multidimensional streams*. In connection to our preliminary work [17], we further plan to investigate modeling of moving objects, introducing algebraic constructs for space-based windows and developing operators for typical spatiotemporal queries, such as range or nearest-neighbor search. Finally, shared execution of various spatiotemporal predicates and window subsumption in multiple dimensions are considered most challenging issues in such a dynamic setting.

References

[1] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120-139, August 2003.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 2006 (to appear).

[4] A. Ayad and J. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows over Data Streams. In *ACM SIGMOD*, pp. 419-430, June 2004.

[5] A. Arasu and J. Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *ACM SIGMOD Record*, 33(3):6-12, September 2004.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *ACM PODS*, pp. 1-16, May 2002.

[7] D. Barbarà. The Characterization of Continuous Queries. *International Journal of Cooperative Information Systems*, 8(4): 295-323, December 1999.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, V. Raman, F. Reiss, and M.A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, Asilomar, California, January 2003.

[9] U. Dayal, N. Goodman, and R.H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. In *ACM PODS*, pp.117-123, March 1982.

[10] L. Golab and M. Tamer Özsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *ACM SIGMOD*, pp. 658-669, June 2005.

[11] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. Efficient Execution of Sliding Window Queries over Data Streams. *Technical Report* CSD-TR-03-035, Purdue University, 2003.

[12] C.S. Jensen and R.T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1): 36-44, January 1999.

[13] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A Heartbeat Mechanism and its Application in Gigascope. In *VLDB*, pp. 1079-1088, September 2005.

[14] J. Krämer and B. Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *COMAD*, pp. 70-82, January 2005.

[15] J. Li, D. Maier, K. Tufte, V. Papadimos and P.A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *ACM SIGMOD*, pp. 311-322, June 2005.

[16] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*, pp. 37-52, January 2005.

[17] K. Patroumpas and T. Sellis. Managing Trajectories of Moving Objects as Data Streams. In *STDBM*, pp. 41-48, August 2004.

[18] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *ICDE*, pp. 232-239, March 1995.

[19] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-only Databases. In *ACM SIGMOD*, pp. 321-330, June 1992.

[20] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3): 555-568, May 2003.