Maintaining Consistent Results of Continuous Queries under Diverse Window Specifications*

Kostas Patroumpas[1] and Timos Sellis[1,2]

[1] School of Electrical and Computer Engineering,
National Technical University of Athens, Hellas
[2] Institute for the Management of Information Systems,
Research Center "Athena", Hellas
{kpatro,timos}@dbnet.ece.ntua.gr

Abstract. Continuous queries applied over nonterminating data streams usually specify windows in order to obtain an evolving –yet restricted– set of tuples and thus provide timely and incremental results. Although sliding windows get frequently employed in many user requests, additional types like partitioned or landmark windows are also available in stream processing engines. In this paper, we set out to study the existence of *monotonic-related* semantics for a rich set of windowing constructs in order to facilitate a more efficient maintenance of their changing contents. After laying out a formal foundation for expressing windowed queries, we investigate update patterns observed in most common window variants as well as their impact on adaptations of typical operators (like windowed join, union or aggregation), thus offering more insight towards design and implementation of stream processing mechanisms. Furthermore, we identify syntactic equivalences in algebraic expressions involving windows, to the potential benefit of query optimizations. Finally, this framework is validated for several windowed operations against streaming datasets with simulations at diverse arrival rates and window specifications, providing concrete evidence of its significance.

1 Introduction

Continuous queries over data streams should provide near real-time incremental response in mission critical applications, such as telecom fraud detection, stock exchange bids or traffic surveillance systems. Stream processing must keep up with the fluctuating rate of high-volume transient items arriving from numerous sources (routers, sensors etc.), otherwise dropping unprocessed tuples is inevitable [27]. It cannot be expected that fast in-memory computation could be performed over the entire stream "history", lest that available system resources would get rapidly exhausted. Besides, due to the incessant arrival of fresh data, the significance of each isolated item is time-decaying for most realistic applications. Thus, a common practice is to limit the set of stream items considered in computation, using techniques such as punctuations [30], synopses [7], but most

* This paper appears in *Information Systems*, 36(1): 42-61, March 2011.

predominantly, window primitives already introduced for OLAP functions with the SQL:1999 standard [22].

Windows are abstractions specified through distinctive properties inherent in the incoming data, aiming to provide finite stream portions for efficient query processing. Typically, users submit queries that specify a particular period or stream items of interest, such as "*every minute* find the average temperature over readings collected *during the past hour*". In Continuous Query Language (CQL) [4], this request can be expressed as follows:

```
Q1: SELECT AVG(Temperature)
    FROM Readings [RANGE 60 MINUTES SLIDE 1 MINUTE]
```

In effect, a *time-based sliding* window is repetitively applied over the stream (every minute in the example above) and returns a temporary relation containing its most recent items at any given time, e.g., tuples received in the past hour. Afterwards, other operators specified with the query (e.g., aggregation in Q1), should be reevaluated against each instance of the temporary relation. Certain other types, including count-based, partitioned, tumbling or landmark windows, have also been suggested for specifying suitable stream subsets. Several research prototypes, such as AURORA [1], Borealis [2], Gigascope [17], STREAM [7], and TelegraphCQ [11] have been implemented, each one offering a variety of windows. This development has paved the way to emerging commercial systems, like StreamBase [28], Coral8 [3] or Oracle's Complex Event Processor [23], eventually setting a foundation for large-scale Stream Processing Engines [27].

The contents of a window at any given time represent its *state* and change continuously following the evolving data stream. As it becomes apparent from the sliding window example, fresh items get included into the window state, while older elements expire since they arrived more than an hour ago. For typical windowing constructs, stream tuples may participate in multiple successive states, but each state does not necessarily subsume its preceding one. That's why a continuous query involving windows is characterized *non-monotonic*; as we explain later, its results cannot be produced in an incremental fashion due to data expirations in the underlying window(s).

Windows can be usually specified either according to the ordering of streaming items (*tuple-based*) or the succession of their associated time indications (*time-based*) [24]. Thanks to such *timestamps* that control admission and expiration of tuples from the current state, many window types exhibit interesting repetitive patterns when refreshing their contents. For instance, an item that has just been inserted into an one-hour long sliding window, will be surely discarded after exactly one hour. Although this key observation cannot entirely eliminate the burden of non-monotonicity in windowing constructs, it may still facilitate efficient state maintenance.

Moreover, window update patterns influence the validity of results for adjoined relational operators, such as aggregations or joins over windows. In the aforementioned example, the average value should be renewed every minute, since aggregation is controlled by the sliding window. However, costly recomputation against the entire window could be avoided, by only considering newly

inserted and discarded items. As we discuss in Section 4, many windowed operators display update characteristics that stem from operation semantics but also depend on the type of window.

In this work, we conduct a careful examination of window semantics and properties with respect to *monotonicity*, extending a previous analysis focusing strictly on update patterns of time-based sliding windows [14]. We discuss in more detail our ideas from [25] and develop heuristics that can be proven valuable to continuous query evaluation. To the best of our knowledge, ours is the first framework that covers all major window variants and operators, also offering a concrete insight of their intrinsic characteristics. We stress that our focus is on properties of individual operators, and not on complete query execution plans that would raise issues beyond the scope of this paper (such as operator scheduling, data propagation and query plan optimization). In that respect, our work is orthogonal to recent efforts towards development of stream processing engines. Overall, we consider windows as first-class citizens in stream processing, fully integrated into a powerful set of operators that can express a wide range of continuous queries.

In summary, the main contributions of our work are:

- We consolidate a formal framework for expressing continuous queries, by means of a model that captures the evolving features of data streams and also adheres to well-founded relational concepts.
- We provide a classification of most typical window variants and discuss how inherent update patterns –although not strictly monotonic– may lead to nearly smooth maintenance of their changing state.
- We present windowed adaptations of standard relational operators employed in continuous queries and we investigate means to overcome certain implications caused from window update characteristics.
- As an essential step towards a stream algebra, we outline syntactic equivalences in algebraic expressions involving windows.
- Finally, we validate this framework against streaming datasets at assorted arrival rates and demonstrate its benefits in evaluation of autonomous query operators with a variety of window specifications.

The remainder of this paper is organized as follows. In Section 2, we outline essential notions on data streams and continuous queries. Section 3 develops monotonic-related semantics inherent in typical windowing constructs. In Section 4, we discuss the effect of window update patterns on relational operators. Section 5 presents certain algebraic laws affecting syntactic equivalences in windowed operations. Results from an experimental validation are reported in Section 6. Section 7 surveys recent work on stream processing relating to windows, whereas Section 8 concludes the paper.

2 Towards a Stream Processing Framework

In this section, we provide a formalization of stream items from a relational perspective, also incorporating crucial notions like timestamping and ordering.

We outline the main principles concerning semantics of continuous queries over data streams and discuss subtleties caused from possible updates in emitted query results.

2.1 A Relational Representation for Stream Items

Data streams can be generated from readings collected from sensors, events emitted from RFID's or bar code scanners, traffic measurements in transportation systems, signals from network routers etc. Such streaming items are commonly represented as relational tuples [4], not excluding a semistructured form [9, 21]. 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 $\mathcal{A}_i$ and its values are drawn from a possibly infinite atomic* data type domain *$\mathcal{D}_i$. Every tuple is an instance of the schema and it is described by its values at the respective attributes.*

Yet, stream items are something more than relational tuples; *ordering* must be established among such interminably flowing data. Hence, a timestamp value is usually assigned to every streaming tuple, either at its source (e.g., sensors mark their readings) or upon admission to the system. This value is often a time indication like a clock tick from a global chronometer (say, with a granularity of seconds), so data elements generated simultaneously or arriving synchronously get identical timestamp values. Alternatively, simple sequence numbers may also serve as a means of ordering tuples, i.e., a unique serial number is attached to every incoming item. The following definition covers both interpretations:

Definition 2. Time Domain *$\mathbb{T}$ is 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 regarded 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 distinct time instants occurring between its bounds. A typical assumption [4] is that at each timestamp $\tau \in \mathbb{T}$, a possibly large, but always finite number of data elements are admitted for processing. Besides, multiset (bag) semantics apply, signifying that zero, one or multiple identical tuples may be arriving at any single instant:

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

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:

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

On the other hand, the *instance* of a stream at any distinct time moment is a finite multiset of tuples with that specific timestamp value:

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 $S_I(\tau_i) = \{s \in S : \ s.\mathcal{A}_\tau = \tau_i\}$.*

Timestamps offer a unique time indication for each item, and also establish a common time reference among all incoming tuples. Each tuple maps to exactly one timestamp, although 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 $f_O : \mathcal{D}_S \rightarrow \mathbb{T}$ from data type domain $\mathcal{D}_S$ 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* $s_1.\mathcal{A}_\tau \leq s_2.\mathcal{A}_\tau$, *then* $f_O(s_1) \leq f_O(s_2)$.

Temporal ordering is crucial for a streaming mechanism because data items must be given for processing in accordance with their timestamps. As a general rule when evaluating a continuous query at time τ, all stream tuples with timestamps up to that particular τ must be available. Hence, no incoming item should propagate for evaluation if its timestamp value is less than the latest tuple produced by the system. In the sequel, we assume that input tuples are always received in timestamp order, so we do not consider stream imperfections such as delayed or out-of-order items.

2.2 The Importance of Querying Continuously

Compared to one-time queries [7] 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. 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 current stream contents at every distinct time instant. Similarly to [8, 29], we can formally define:

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

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

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 against disjoint stream portions is no better: if only intermediate stream contents are considered in each evaluation, it may happen that newer results contradict formerly given answers. Consider the case of eliminating duplicates from such "delta" stream chunks: although distinct items are correctly obtained from each portion, duplicates could still exist after unifying partial results.

A conservative approach is to accept continuous queries with *append-only results*, thus not allowing any deletions or modifications at already produced answers. For instance, sensor readings indicating temperatures $\geq 30^o$C can be immediately identified as they flow into the system:

```
Q2: SELECT * FROM Readings
    WHERE Temperature >= 30
```

Qualifying items are appended one by one to the result of query Q2, while returned answers always remain valid thereafter. This class of continuous queries is called *monotonic* [8]:

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

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

where $\mathcal{Q}(S(\tau_i))$ denotes results for query $\mathcal{Q}$ 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. In [30] a characterization of query operators is introduced, according to their suitability for stream processing. *Stateless* filtering operators (like projections or selections) never delete tuples already emitted in their output stream, hence their incremental evaluation is possible and no past items need be retained as a "state" from transient stream contents. But *blocking operators*, like aggregation or sorting, cannot produce even a single tuple of their result before consuming the entire input. Besides, *stateful operators*, such as join or intersection, may involve stream items that have arrived at previous time instants, so an unbounded state should be continuously maintained for them. Indeed, in order to execute a join, all tuples from both streams must be kept in memory, just in case a newly arriving data item matches an older tuple from the other stream.

The exact role of relational tables in continuous queries is another concern. Whereas in Gigascope [17] 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 [4]). In the latter case, insertion and deletion tuples are used to represent the changing state of such a relation. In [14] 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 their interaction with relations.

2.3 Monotonic-related Classification of Continuous Queries

It is important to note that monotonicity refers to query results and not to incoming stream items. In that context, not only stateless, but also several stateful operators (e.g., join) evaluated against current stream contents $S(\tau)$ provide monotonic results. However, it cannot be always expected that system resources suffice for the growing demands of such unbounded datasets. As already mentioned, in order to bound the increasing memory requirements of query operators, sliding windows are usually applied over the infinite streams to return a finite set with the most recent data items.

Nevertheless, when applying windows over streams, *non-monotonic* results are generally returned. For instance, at each instantiation of a sliding window over a stream, several new tuples get produced, but at the same time some older items expire due to window movement (Fig. 1a). As we explain later, this phenomenon may be aggravated in case of operators specified over windows, like windowed join or difference. As a possible remedy for certain non-monotonic operators, *expiration timestamps* [14, 26] may be assigned to results at another attribute $\mathcal{A}_{exp}$ denoting their validity interval. This can be advantageous, since each no longer valid output tuple can be eliminated without inspecting again any input or intermediate data.

Another interesting technique for evaluating sliding window queries is to introduce *negative tuples* [13] as a means of cancelling previously emitted, but outdated results. In general, non-monotonic queries may have results that expire at unpredictable times; in case of an expiring item p, a negative tuple p^- must be generated as an artificial copy of p, so as to signify its removal from the result. Apart from doubling the volume of tuples flowing through the system, this policy also entails reengineering of the query evaluation process, since operators should be considerably enhanced in order to enable handling of both regular (positive) and cancellation (negative) tuples. According to [14], the negative approach is not generally recommended, but usage of negative tuples cannot be ruled out entirely, because in certain cases they prove indispensable for expunging unpredictably expired results.

A meticulous study on monotonicity of query operators coupled with time-based sliding windows has been presented in [14], also proposing a classification of such continuous queries according to their update patterns. Besides strictly *monotonic* and *non-monotonic* operators, two intermediate categories were suggested:

- In *weakest non-monotonic* operators, results get appended to and discarded from the output in a first-in-first-out (FIFO) fashion. For instance, when a sliding window is applied over a stream, for each item currently in that frame we know beforehand when it will be discarded. Therefore, an expiration timestamp can be calculated for each incoming tuple marking its validity within the window.
- *Weak non-monotonic* operators do not generally show a FIFO pattern in the way their results expire, but expiration times can be determined for all results without emitting negative tuples.

In this work, we adopt this characterization and develop it further to cover more than time-based sliding window queries. We also generalize the notion of expiration beyond its usual temporal semantics: we introduce *expiration order* to appropriately handle the case of tuple-based window variants and operations. We demonstrate that such a monotonic-related framework can also explain the behavior of major window variants and facilitate evaluation strategies for many problematic operators.

3 Semantics and Maintenance of Windows

In this section, we focus on precise specification of windows where typical operators may be applied under well-known relational semantics. We think that our proposition for window semantics is flexible enough to be used under any of the aforementioned interpretations of continuous queries.

3.1 Specifying Windows over Data Streams

In most stream processing engines, submission of a continuous query is always accompanied by –mostly sliding– window specifications on any stream involved in that query. A window is a *stream-to-relation* operator [4] that applies flexible bounds on the unbounded stream in order to fetch a finite, yet ever-changing collection of tuples, which may be regarded as a temporary relation. Formally:

Definition 9 (Window over Data Stream). *Let $\mathcal{W}_P$ a window with conjunctive condition $\mathcal{C}_P$ 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:*

$$\forall\ \tau_i \in \mathbb{T},\ \tau_i \geq \tau_0,\ \mathcal{W}_P(S(\tau_i)) = \{s \in S(\tau_i) :\ \mathcal{C}_P(s, \tau_i)\ holds\}$$

provided that $|\mathcal{W}_P(S(\tau_i))| \leq n$, for any large, but always finite $n \in \mathbb{N}$. Parametric set P constitutes the signature *of $\mathcal{W}_P$ with its inherent properties.*

Therefore, each window is applied over the items of a single data stream S and at every τ_i returns a concrete finite multiset of tuples $\mathcal{W}_P(S(\tau_i)) \subset S(\tau_i)$ which is called the *window state* at this time instant. When a continuous query involves multiple streams (e.g., joins), a separate windowing construct must be specified for each one, even if identical semantics are applied to all of them (i.e., similar expressions $\mathcal{C}_P$). In contrast to SQL:1999 specifications for OLAP functions [22], these windows always look "backward" to *preceding* stream contents and never "forward" to items *following* in the future; naturally, it cannot be known in advance the admission time or arrival rate of tuples in the time-varying flow of the unbounded stream.

Essentially, the exact window structure is determined through a signature P that prescribes its distinctive properties: (i) its time-varying *lower* and *upper bounds*, (ii) a window *extent* ("size"), and (iii) its adjustment or progression across time. Condition $\mathcal{C}_P$ involves a monotonically-increasing *windowing*

attribute [21] that is being checked for qualifying tuples. Since we adhere to timestamps for establishing order among stream elements, we designate $\mathcal{A}_\tau$ as the windowing attribute for condition $\mathcal{C}_P$. As we present next, such conjunctive conditions can be expressed with a family of *scope functions*, each one appropriate for a specific window variant. Typically, such a function takes as arguments a time interval of interest or a specific tuple count and each time returns the actual window state (e.g., all tuples received during the past hour or the most recent 100 tuples).

Assuming one second as the timestamp unit, the sliding window signature for query Q1 includes: time τ_0 the frame is initially applied, a fixed size in time units (3600 seconds) and a sliding step that periodically moves both bounds forward (every 60 seconds). Consequently, window state at each τ_i is finite: $\mathcal{W}_P(S(\tau_i)) = \{s \in S(\tau_i) : \ s.\mathcal{A}_\tau > 60 \cdot \lfloor \frac{\tau_i}{60} \rfloor - 3600 \ \wedge \ s.\mathcal{A}_\tau \leq 60 \cdot \lfloor \frac{\tau_i}{60} \rfloor\}$.

Besides, conjunctive condition $\mathcal{C}_P$ may be extended with additional filtering predicates on other attributes apart from the windowing one, in a way that *value-based* [7] or *predicate-based* windows [12, 23] may be expressed as well. Next, we attempt a rigorous algebraic description of the principal window types proposed in the context of data streams, using timestamps as windowing attributes. We adopt from [7] the basic discrimination in *logical* and *physical windows*, according to the unit (time interval or tuple count, respectively) used to determine window contents. A taxonomy of window types and a detailed discussion about their properties can be found in [24].

3.2 Logical Window Variants

In logical windows, the timestamp values of streaming tuples are checked for inclusion within a time interval. We conveniently express this requirement by means of a *scope function*, which 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, such a function returns the current window bounds and *not* actual stream items, taking as parameters the signature properties of the respective window type (extent, progression step, etc.).

In the following, we present the most representative variants of logical windows that have been implemented for several stream engines, although the expressiveness of this approach has a broader applicability.

Time-based Sliding Windows. This windowing construct is perhaps the most widely used in continuous queries over streams. Let $\tau_0 \in \mathbb{T}$ be the time instant that a continuous query is initially submitted specifying a sliding window. Usually, the scope of such a window is defined with a fixed-size temporal extent ω signifying the most recent time interval (e.g., "continuously return all stream items arrived during the past hour"), as well as a progression step β that denotes

how often the state should be refreshed (e.g., "slide every minute"). Formally:

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

In [24] we provide a more general definition considering the case that the upper bound of the window has a delay (or *lag*) δ with regard to the current time instant τ. But usually, the upper bound of the sliding frame coincides with the current timestamp of the stream (i.e., $\delta = 0$). 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 $\tau \in \mathbb{T}$, the qualifying tuples are included in the window state:

$$\mathcal{W}_{\langle \tau_0,\omega,\beta \rangle}(S(\tau)) = \left\{ s \in S(\tau) :\ s.\mathcal{A}_\tau \in scope_{\langle \tau_0,\omega,\beta \rangle}(\tau) \right\}$$

by appending any newly arrived tuples and discarding older ones on the basis of their time indications. Stream arrival rates may fluctuate, so a different number of items can be returned at any given time (Fig. 1a).

In CQL [4, 23], a clause like `S[RANGE` ω `SLIDE` β`]` is used to declare a time-based sliding window over stream S. By setting $\tau =$ `NOW`, $\omega = 1$, and $\beta = 1$ in the scope function, it is also straightforward to express an important subclass of sliding windows that obtain the current instance $S_I(\tau)$ of the stream. In CQL, the window shortcut $S\,$`[NOW]` is used to get all tuples from stream S with the current timestamp value.

In the most general case where $\beta < \omega$, overlaps are observed between any two successive states of a sliding window, thus a subset of their contents remains intact across states (common tuples for both ω in Fig. 1a). Window states are reevaluated every β time units; in the meantime, 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 bounds change discontinuously at time instants that depend strictly on the pattern stipulated by the progression step β. The first branch 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. Henceforth, both window bounds move in unison with the evolution of time, always spanning a fixed interval ω.

Progression step β could be set equal to the finest time granule (e.g., one second), so that the window slides smoothly in pace with the advancement of time. In that case, the recursive branch in the aforementioned scope function is redundant, as window's contents are modified at every time instant. Typically, the window may slide at a unit step $\beta = 1$ (i.e., every instant), although a multi-hop progression step $1 < \beta < \omega$ is also an option.

Depending on progression step β between successive evaluations at times τ_{i-1} and τ_i, a varying number of common tuples may be found in the corresponding states (Fig. 1a). It can be easily proven the following

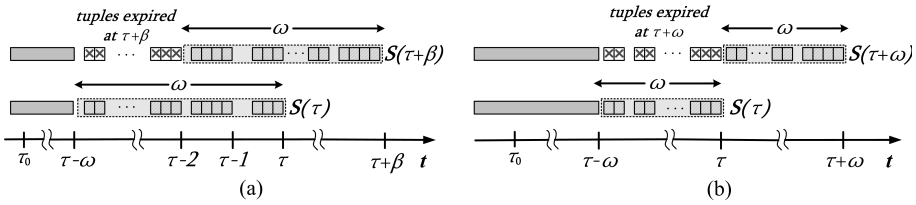

Fig. 1. Window states for time-based constructs at two instants τ and $\tau + \beta \in \mathbb{T}$. (a) Successive states of a *time-based sliding* window with extent ω and sliding step $\beta < \omega$. (b) *Tumbling* window with $\beta = \omega$ and consecutive states with no overlapping tuples.

Proposition 1. *Between successive states of a time-based sliding window with $\beta < \omega$, assuming that $\tau_i = \tau_{i-1} + \beta$, it holds that:*

$$| \mathcal{W}_{\langle \tau_0, \omega, \beta \rangle}(S(\tau_{i-1})) \bigcap \mathcal{W}_{\langle \tau_0, \omega, \beta \rangle}(S(\tau_i)) | \geq 0.$$

In fact, new tuples may be appended to the current window state, while evicting a possibly different number of older items, since the count of tuples arriving at each timestamp may vary. But for each newly qualifying tuple, we can easily predict when it will be permanently removed from state, due to the always fixed ω. Indeed, an expiration timestamp τ^{exp} can be attached to every tuple s with timestamp τ, once s arrives and gets included in the window state. Given an extent ω, it is evident that at time $\tau^{exp} = \tau + \omega$ this tuple s will be discarded from state.

Despite such continuous change in window states, ordering among qualifying tuples is always preserved, since items are included into and excluded from the sliding window in a *first-in-first-out* (FIFO) fashion. For this reason, time-based sliding windows are characterized as *weakest non-monotonic* in [14]. Indeed, if τ, τ^{exp} respectively denote the original timestamp value and expiration time of a qualifying stream tuple s, then:

Proposition 2. *For state $\mathcal{W}_{\langle \tau_0, \omega, \beta \rangle}(S(\tau_i))$ at time $\tau_i \in \mathbb{T}$ of a time-based sliding window over data stream S, it holds that:*

$$\forall\ s_1, s_2 \in \mathcal{W}_{\langle \tau_0, \omega, \beta \rangle}(S(\tau_i)),\ s_1.\tau \leq s_2.\tau \Leftrightarrow s_1.\tau^{exp} \leq s_2.\tau^{exp}.$$

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 disjoint "batches" that span a given 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 [17]. In case that $\beta = \omega$, window states are disjoint but consecutive: the lower bound of the current state and the upper bound of its preceding one are successive time instants. For example, average network traffic may be computed every $\beta = 30$ minutes, considering all packets transferred within the past half hour ($\omega = 30$

minutes). At each evaluation, separate stream portions are returned and thus window contents are obtained in a discontinuous fashion:

$$\mathcal{W}_{\langle \tau_0,\omega,\omega\rangle}(S(\tau)) = \{s \in S(\tau) :\ s.\mathcal{A}_\tau \in \mathit{scope}_{\langle \tau_0,\omega,\omega\rangle}(\tau)\}$$

Alternatively, for several applications (e.g., traffic monitoring), different window sizes might be needed (e.g., for peak or night hours, weekends etc.). Even then, function $\mathit{scope}_{\langle \tau_0,\omega,\beta\rangle}$ is still valid by setting a time-varying extent $\omega(\tau)$. In [1] tumbling windows may be accompanied with user-defined predicates for identifying the end of temporary states, but this approach is mainly geared towards implementation efficiency rather than query semantics.

As prescribed by tumbling window semantics, every state ceases to exist in its entirety upon initiation of its succeeding one (Fig. 1b). Hence:

Proposition 3. *No common items occur between successive states of a tumbling window at time instants* τ_{i-1} *and* $\tau_i \in \mathbb{T}$*:*

$$\mathcal{W}_{\langle \tau_0,\omega,\omega\rangle}(S(\tau_{i-1})) \bigcap \mathcal{W}_{\langle \tau_0,\omega,\omega\rangle}(S(\tau_i)) = \emptyset.$$

Despite appearances, discussion about monotonicity for a tumbling window should not be ruled out altogether. Expirations occur every β units and are known beforehand; for the k-th state of a window specified at τ_0, a single expiration time $\tau^{exp} = \tau_0 + k \cdot \beta$ is computed. During that state, tuples continue to pile up without removing previous items. Since the contents of each state grow monotonically until their simultaneous elimination, a tumbling window is a *weak non-monotonic* operator with known expiration times.

Meanwhile, in terms of efficient maintenance, a tumbling construct may be loosely considered as a sliding one with the same extent; its bounds shift in unison at each time unit (e.g., every second or at each new timestamp value). The only difference is that a tumbling window discloses its state periodically, as specified by its progression step β. A practical evaluation policy would be to remove a tuple participating in the current state at the same order it was originally inserted, emulating a sliding window pattern with some kind of deferred elimination of tuples in plain FIFO fashion.

Landmark Windows. Such windows maintain one of their bounds fixed at a specific time instant τ_ℓ (*"landmark"*), letting the other follow the evolution of time. Most applicable is the case of a *lower-bounded landmark* window, having a permanent starting time τ_l for all its frames, whereas its upper bound varies with time (e.g., "get all recordings collected after 10 p.m."). If this construct is applied at time τ_0, then at any subsequent $\tau \geq \tau_0 \in \mathbb{T}$ the scope function takes the form:

$$\mathit{scope}_{\langle \tau_0,\tau_l\rangle}(\tau) = \begin{cases} \emptyset & \text{if } \tau_0 \leq \tau < \tau_l \\ [\tau_l,\ \tau] & \text{if } \tau_0 \leq \tau_l \leq \tau \end{cases}$$

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

$$\mathcal{W}_{\langle \tau_0,\tau_l\rangle}(S(\tau)) = \{s \in S(\tau) :\ s.\mathcal{A}_\tau \in \mathit{scope}_{\langle \tau_0,\tau_l\rangle}(\tau)\}$$

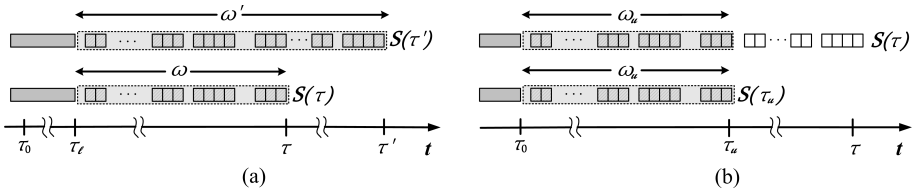

Fig. 2. Window states for landmark constructs. (a) *Lower-bounded* window with growing state after time τ_ℓ. (b) *Upper-bounded* window has a frozen state as soon as $\tau > \tau_u$.

This window type will never discard qualifying tuples while it keeps appending new items indefinitely, unless either the query is explicitly revoked (i.e., the window is cancelled) or the stream is exhausted and no tuples enter into the system anymore. Thus, such windows are potentially unbounded.

Contrary to other window types, lower-bounded landmark windows are strictly *monotonic*, because no tuple ever expires from any window state. At each time instant, the current state clearly subsumes its previous ones, practically containing all their tuples (Fig. 2a). Hence:

Proposition 4. *For a landmark window with a lower-bound at time* $\tau_l \in \mathbb{T}$*, it holds that:* $\forall\ \tau', \tau'' \in \mathbb{T},\ \tau_l \leq \tau' \leq \tau'',\ \mathcal{W}_{\langle \tau_0, \tau_l \rangle}(S(\tau')) \subseteq \mathcal{W}_{\langle \tau_0, \tau_l \rangle}(S(\tau''))$.

Less commonly, an *upper-bounded landmark* window retains a preset upper edge, most usually a future time instant τ_u. Initially, window state keeps expanding, but once τ_u is reached the window will "close" and its state remains "frozen" thereafter (Fig. 2b). In the rare case that both bounds are static, a *fixed-band window* is defined, specifying a rigid time interval as a constraint on timestamp values. It can be trivially proven that both types are also monotonic. Please refer to [24] for further details on these variants.

3.3 Physical Window Variants

Since physical windows are determined by a predefined number of tuples and their extent spans the most recent stream elements, they are sliding by default. In the following, we assume a unary progression step, 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 fetches the $n \in \mathbb{N}^*$ most recent tuples from a stream S:

$$\mathcal{W}_{\langle n \rangle}(S(\tau)) = \{s \in S(\tau) : \exists\ \tau' \in \mathbb{T}\ (\tau' \leq \tau \wedge |\{s \in S(\tau) : s.\mathcal{A}_\tau \in [\tau', \tau]\}| \leq n)$$
$$\wedge\ \forall\ \tau'' \in \mathbb{T}\ (\tau'' < \tau'\ \wedge |\{s \in S(\tau) : s.\mathcal{A}_\tau \in [\tau'',\ \tau]\}| > n) \wedge s.\mathcal{A}_\tau \in [\tau',\ \tau]\}.$$

Overlapping states may occur between successive evaluations, as illustrated in Fig. 3a. Intuitively, starting from current time instant τ and going steadily

backwards within changeable scope $[\tau', \tau]$, qualifying tuples are obtained until their total count does not exceed threshold n. Nevertheless, subtle issues may arise with this policy. When the contents of count-based windows are derived through their sequence numbers [4], it must be clear how many times possible duplicates are counted and how ties are broken for the n-th element. A similar case concerns tuples with time indications: ties may still occur at the lower window bound, when only k items should be chosen out of a batch of $m > k$ tuples at τ' 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 [4].

In practical implementations, such a sliding fixed-count extent is accomplished by discarding the most remote item from current window state so as to accept each newly arrived one. Implicitly, such a policy enumerates each qualifying tuple with a monotonically increasing *admission order* $\rho \in \mathbb{N}$, and also assigns an *expiration order* $\rho^{exp} = \rho + n$. Clearly, $\rho = \tau$ for every item when sequence numbers serve as timestamps. A data structure such as a queue of fixed size n is sufficient to accommodate a window state with FIFO update patterns. Thus, count-based sliding windows are characterized as *weakest non-monotonic*, exactly like their time-based counterparts, given that the ordering of expirations is actually equivalent to shifting by n the original admission numberings:

Proposition 5. *For state $\mathcal{W}_{\langle n \rangle}(S(\tau_i))$ at time $\tau_i \in \mathbb{T}$ of a count-based sliding window over data stream S, it holds that:*

$$\forall\ s_1, s_2 \in \mathcal{W}_{\langle n \rangle}(S(\tau_i)),\ s_1.\rho \leq s_2.\rho \Leftrightarrow s_1.\rho^{exp} \leq s_2.\rho^{exp}.$$

By relaxing the notion of expiration time, our interpretation of update patterns with respect to physical windows actually broadens the initial proposal in [14]. Indeed, for a time-based sliding window, it is always possible to predict when a tuple will be evicted from state, by calculating *absolute* expiration timestamps with explicit time indications. In contrast, a given tuple s partaking in a count-based window will be discarded *relatively* to the amount of items succeeding s in the stream. Exact expiration time of s depends on the actual (and possibly fluctuating) stream rate, but expiration order is certain: tuple s can be removed as soon as n items have been accepted after s. In the sequel, unless otherwise specified, the notion of expiration time covers both aspects, i.e., absolute time indications and relative orderings.

Partitioned Windows. This demultiplexing operator implies that the incoming items of stream S are subdivided into several substreams $S_1, S_2, \ldots$ according to their values at certain *grouping attributes* $\mathcal{L} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\}$, excluding the windowing attribute $\mathcal{A}_\tau$. For every combination of values $g_i = \langle a_1^i, a_2^i, \ldots, a_k^i \rangle,\ a_j^i \in \mathcal{D}_j$ identified on the respective attributes of $\mathcal{L}$, a new partition is created:

$$S_i(\tau) = \{s \in S(\tau) : \forall \mathcal{A}_k \in \mathcal{L},\ a_k^i \in g_i,\ s.\mathcal{A}_k = a_k^i\}.$$

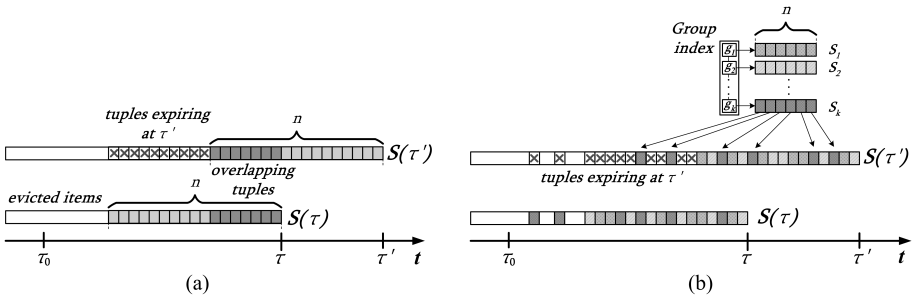

Fig. 3. Window states for tuple-based variants at two instants τ and τ'. (a) *Count-based sliding* window of size n tuples. (b) *Partitioned* window with n tuples at each partition.

Then, at each time instant $\tau \in \mathbb{T}$, the n most recent tuples are taken from each substream S_i corresponding to a distinct *group identifier* g_i as its contribution to the overall window state. Essentially, a separate count-based window is applied over each S_i and the current state is derived as the union of these partial states:

$$\mathcal{W}_{\langle \mathcal{L},n \rangle}(S(\tau)) = \bigcup_{g_i} \mathcal{W}_{\langle n \rangle}(S_i(\tau))$$

For example, to identify current trends for a financial application, analytics can be computed each time against the 1000 most recent stock readings from each particular sector, by distinguishing into industrial, banking, telecom etc. stocks with a `PARTITION BY` clause [4, 23]. Apparently, *count-based windows* can be regarded as a special case of partitioned ones, since no grouping attributes are specified and the stream itself is the only partition. Note that the state of a partitioned window is a multiset of stream items, without involving aggregation after grouping data elements.

As shown in Fig. 3b, a partitioned window essentially applies a separate sliding frame against each of the partial substreams $S_1, S_2, \ldots$ in which the original stream S is being demultiplexed. Therefore, the tuples in each of the n-sized constituent partitions expire in FIFO order. Still, it should not be expected that the obtained tuples change at the same regular rate for each partition, because it may occur that some partition S_i has much older tuples compared to another partition S_j. Some combinations of values (e.g., those defining S_j) on the grouping attributes may be observed frequently while some others less often (e.g., for S_i), depending on the actual patterns detected in the incoming tuples. It is also possible that during some time interval no fresh tuples qualify for a partition, which still retains its state as created from items received long ago.

Since overall window state is always obtained from the union of all partitions, it turns out that expiration order of items does not generally coincide to their succession in timestamp values, not even to their insertion order into the window state. Each arriving item may cause a deletion at the partition it belongs to,

therefore it may alter ordering at a non-specific position of the entire window state. It is also clear that each incoming tuple must be accepted in the state, so it can be directly appended to the corresponding partition. However, the problem is how to expunge the oldest tuple that belongs to the same partition as the newly inserted item, in order to retain no more than n items from this group.

To resolve this issue and respect ordering in each window state, we introduce a data structure with interconnected autonomous queues illustrated in Fig. 3b. A list indexes all group identifiers with each g_i reflecting a partition S_i observed thus far; note that a partition will be present forever as soon as that combination of attribute values has been identified in the streaming tuples. A separate queue is maintained for each partition S_i, with all its items sorted by their expiration order ρ_i^{exp} regarding S_i. Since each node points to the respective item at the window state, only local arrangements in pointers (one insertion and one deletion) are needed for each incoming tuple instead of state reconstruction from scratch. By maintaining each partition separately, their update pattern resembles to that of count-based windows. Accordingly, partitioned windows can be considered as *weak non-monotonic*, and do not necessitate generation of negative tuples.

4 Identifying Update Patterns in Windowed Operators

The main motivation behind introduction of windows is the necessity to unblock query operators in stream processing. But, when windows are intertwined with typical query operators, each variant presents its own challenges with respect to evaluation. The crucial differentiation with relational semantics is that results of windowed analogues must be continuously renewed, keeping in pace with possibly unpredicted changes in window state(s). For example, a windowed join should check for matching tuples between evolving windows applied over its streaming sources. Windowed operators accept streams of timestamped tuples as input and generate temporary relations as output. If resulting items must be reassembled as a stream for further processing, a converse *relation-to-stream* operator is used (like `ISTREAM`, `DSTREAM`, `RSTREAM` in [4]), assigning timestamp values as well.

In this section, we briefly outline the basic characteristics of common unary and binary operators with precise semantics as derived from our analysis in [24]. Most importantly, we systematically investigate repercussions that arise due to monotonic-related semantics of their associated window frames. Operators are distinguished in *neutral*, *interfering* and *inflexible*, according to their reaction to window update patterns. Binary operators are considered over identical window variants (e.g., two partitioned windows), although dissimilar signatures are allowed (e.g., different tuple count per partition); we defer discussion for diverse window specifications to Section 4.4. Table 1 summarizes the resulting classification, as derived from the following discussion for each windowed operator.

4.1 Neutral Operators

As pointed out in Section 2, specification of windows for *projection* and *selection* is not strictly necessary, because both operators act like filters over each stream-

Table 1. Monotonic-related classification of typical windowed operators.

Window variants	*Monotonic*	*Weakest non-monotonic*	*Weak non-monotonic*	*Non-monotonic*
sliding		$\sigma_{\mathcal{F}}^{W}$, $\pi_{\mathcal{L}}^{W}$, $\underset{W}{\cup}$	δ^{W}, $\underset{W}{\bowtie}$, $\gamma_{\mathcal{L}}^{f\,W}$	$\underline{W}$
tumbling			$\sigma_{\mathcal{F}}^{W}$, $\pi_{\mathcal{L}}^{W}$, $\underset{W}{\cup}$,δ^{W}, $\underset{W}{\bowtie}$, $\gamma_{\mathcal{L}}^{f\,W}$	$\underline{W}$
partitioned			$\sigma_{\mathcal{F}}^{W}$, $\pi_{\mathcal{L}}^{W}$, $\underset{W}{\cup}$,δ^{W}, $\underset{W}{\bowtie}$, $\gamma_{\mathcal{L}}^{f\,W}$	$\underline{W}$
landmark	$\sigma_{\mathcal{F}}^{W}$, $\pi_{\mathcal{L}}^{W}$, $\underset{W}{\cup}$,δ^{W}, $\underset{W}{\bowtie}$, $\gamma_{\mathcal{L}}^{f\,W}$			$\underline{W}$

ing tuple. Besides, binary *union* combines into a single multiset incoming tuples from two (or multiple) sources, essentially merging their current instances $S_I(\tau)$ for each successive τ. Without applying windows, these operators actually work in a *stream-in-stream-out* basis, providing incremental results with no need to maintain their state.

However, in certain circumstances, query semantics either include window specification (e.g., maintain sensor readings over the past hour) or even impose a suitable frame to facilitate query execution (e.g., `NOW`-based windows for unbounded sources [4]). In that case, selection, projection and union should also maintain a state with their temporary results, i.e., tuples obtained from the respective window(s) that qualify to operator semantics (e.g., selection criteria). In summary:

Windowed Projection. After applying a window[3] $\mathcal{W}$ over the contents of a stream S, operator $\pi_{\mathcal{L}}^{W}$ retains only chosen attributes $\mathcal{L} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\}$ at the schema of qualifying tuples:

$$\pi_{\mathcal{L}}^{W}(S(\tau)) = \pi_{\mathcal{L}}\ (\mathcal{W}(S(\tau))) = \{\langle s.\mathcal{A}_1, s.A_2, \ldots, s.\mathcal{A}_k, s.\mathcal{A}_\tau\rangle : s \in \mathcal{W}(S(\tau))\}$$

This "vertical" operator treats the contents of each window state as a typical relation and it simply removes unnecessary attributes.

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

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

Condition $\mathcal{F}$ may involve comparison operators $\theta \in \{=, \neq, <, \leq, >, \geq\}$ or even a conjunction of expressions. Such a "horizontal" operation drops items from window state, but preserves the schema of qualifying tuples.

Windowed Union. As in relational set-theoretic operations, union adheres to multiset semantics, so schema E is extended with a *multiplicity* attribute for keeping count of $k > 0$ identical tuples within each window state. Provided that

[3] For clarity, we henceforth eliminate signature P in window notation.

both streams have matching schemata, even under diverse window specifications against each one, at every $\tau \in \mathbb{T}$ their union is:

$$S_1(\tau) \bigcup_W S_2(\tau) = \{\langle s, \tau', k_1 + k_2\rangle : \forall\ \tau' \in scope_{W1} \cup scope_{W2}, \exists\ k_1, k_2 \in \mathbb{N},$$
$$((\langle s, k_1\rangle \in \mathcal{W}_1(S_1(\tau)) \wedge s.\mathcal{A}_\tau = \tau') \vee (\langle s, k_2\rangle \in \mathcal{W}_2(S_2(\tau)) \wedge s.\mathcal{A}_\tau = \tau'))\}.$$

The definition above slightly revises the one given in [24], prescribing that union is performed for distinct timestamp values of either window scope with the intention of producing ordered results.

Interestingly, for all aforementioned operators, each resulting tuple maintains its original timestamp, and attribute $\mathcal{A}_\tau$ always gets included in the output. Accordingly, their expiration times remain intact, exactly as determined by window semantics; stale tuples that expire from the window, should also expire from the operator state. Suppose that a sliding window of one hour is applied to (originally monotonic) query Q2:

```
[Q2′]: SELECT *
       FROM Readings [RANGE 60 MINUTES SLIDE 1 MINUTE]
       WHERE Temperature >= 30
```

Contrary to Q2, any results for query Q2′ issued more than an hour ago are no longer valid and actually expire in a FIFO fashion. Windowing is responsible for those expirations, while operator semantics have no influence at all. Since sliding windows are weakest non-monotonic, so it is the windowed selection in Q2′ above. Similar conclusions can be drawn for other window types as well. We call selection, projection and union *neutral operators* over windows, because they play no active role in expiration of their own results. Therefore:

Proposition 6. *With respect to monotonicity, windowed analogues of neutral operators retain the same characterization as their associated window(s).*

This pattern is advantageous, since neutral operators need not check back with their respective window(s) before eliminating invalid results and are always able to maintain consistent answers without usage of negative tuples.

4.2 Interfering Operators

In contrast to neutral ones, other operators like duplicate elimination, join or aggregation, may have a serious impact on the update patterns of their results. These operators require window specifications, hence their answers should be consistent with the actual window state(s). However, the update characteristics of window tuples might get *biased* after passing through such operators: insertions and expirations at the output results often occur in a different fashion compared to updates in window contents. As we discuss next, such *interfering* operators demonstrate similar update characteristics:

Proposition 7. *Duplicate elimination, aggregation and join are monotonic over landmark windows, but weak non-monotonic when accompanied with any other windowing constructs.*

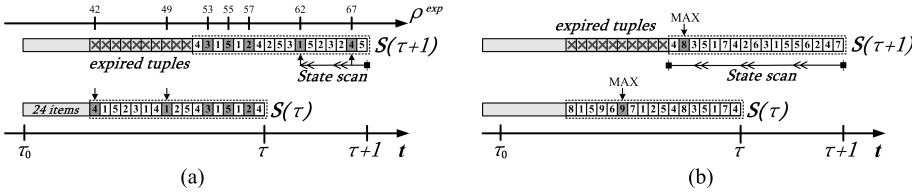

Fig. 4. (a) Handling expired distinct values over a count-based window ($n = 17$ tuples). Each result is assigned an expiration order ρ^{exp} w.r.t. the sequence of window tuples. For instance, distinct value 4 in $S(\tau)$ expires as soon as the 42^{th} tuple is admitted in window (= 24 already expired items + 17 current state items + 1). (b) When aggregate `MAX`=9 expires, it gets replaced after a sequential scan of the entire window state.

For most window types, this phenomenon can lead to a stronger non-monotonic characterization of the combined operator, practically complicating its evaluation. To facilitate processing, next we discuss some simple heuristics specifically for each windowed analogue, but a more sophisticated *indexing of operator states* is a challenging topic for future research.

Windowed Duplicate Elimination. By removing any identical tuples within the current extent of window $\mathcal{W}$ over a stream S, this operator reports the most recent appearance of each distinct tuple:

$$\delta^{W}(S(\tau)) = \delta\ (\mathcal{W}(S(\tau))) = \\ \{s \in \mathcal{W}(S(\tau)) : \nexists\ s' \in \mathcal{W}(S(\tau)), \forall\ \mathcal{A}_i \in E, s'.\mathcal{A}_i = s.\mathcal{A}_i \wedge s'.\mathcal{A}_\tau \geq s.\mathcal{A}_\tau\}.$$

Such an "eager" policy incurs a costly overhead regardless of the type of specified window: each fresh tuple s must be included in the resulting operator state, potentially replacing an older similar tuple s'. But we adopt from [14] a more efficient scheme, by preserving a distinct item s until its expiration from state. The downside is the cost of finding replacements for expiring distinct items, when required (Fig. 4a). This cost is linear with the window state, since the window should be scanned "backwards" starting from its upper bound in order to detect the youngest tuple s identical to the expiring one. The search stops as soon as such s is found; assuming a random distribution of tuple values, s is probably among recent items, hence the state is rarely probed in its entirety.

For landmark windows, we observe that operator δ remains monotonic and with the latter scheme achieves substantial savings: as soon as a distinct tuple is issued, all its subsequent duplicates are simply ignored. With the rest window types, upon expiration, s gets replaced by the most recent identical tuple s'' found in the current window state. Accordingly, tuple s'' may be appended to the answer long after its inclusion in the window state. Evidently, insertion of resulting tuples is affected by the inherent behavior of operator δ and does not generally coincide with the sequence of updates in the respective window. Meanwhile, distinct results retain their expiration time in the output and their deletion is clearly determined by the associated window (Fig. 4a). Hence, for

all but landmark windows, duplicate elimination produces results with known expiration and is classified as *weak non-monotonic*. Interestingly, not even for sliding windows does operator δ obey to a FIFO update pattern.

Windowed Aggregation. As in extended relational algebra, this operator creates groups of window tuples according to their values on $k > 0$ attributes specified in grouping list of $\mathcal{L} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$. Next, for each distinct combination of corresponding values $g_i = \langle a_1^i, \ldots, a_k^i \rangle$, where $a_j^i \in \mathcal{D}_j, j = 1..k$, a function f is applied on the set of values at an *aggregated attribute* $\mathcal{A}_m \notin \mathcal{L}$ taken from all tuples in group g_i. If no attributes are specified, then all tuples in the window are regarded as a single group. Formally:

$$\gamma_{\mathcal{L}}^{f^W}(S(\tau)) = \gamma_{\mathcal{L}}^{f}(\mathcal{W}(S(\tau))) = \{\langle a_1^i, \ldots, a_k^i, f(a_m), \tau \rangle : \exists\ \langle a_1^i, \ldots, a_k^i \rangle, a_j^i \in \mathcal{D}_j, \\ \wedge\ a_m = \{s.\mathcal{A}_m : s \in \mathcal{W}(S(\tau)) \wedge \forall\ j \in \{1, \ldots, k\}, \mathcal{A}_j \in \mathcal{L},\ s.\mathcal{A}_j = a_j^i\}\}.$$

This operator computes a single scalar value for each group g_i by means of typical aggregation functions such as `COUNT`, `SUM`, `MIN`, `MAX` or `AVG` applied over a set a_m of attribute values, rather than creating disjoint partitions of original stream items like a partitioning window (Section 3.3.2). At each evaluation, resulting tuples substitute previous values for each group identifier g_i, and all get identical timestamps τ (e.g., the current time indication in case a global clock exists). Therefore, the result can be materialized as a relational table $\mathcal{M}$ with one row per group.

To facilitate assignment of incoming tuples into groups according to their observed g_i's, we build a hash table $\mathcal{H}$ over the current window state with these group identifiers as keys. As soon as a stream item arrives, we compose an identifier g_i from its values at the specified attributes in $\mathcal{L}$. Clearly, that g_i is hashed into a bucket of $\mathcal{H}$ where all items of the same group are also maintained. Besides, when a tuple expires from state, its removal from the hash table can be performed in a similar fashion.

But for how long can each scalar aggregate be considered valid? Note that some groups may be identified more frequently in the incoming stream, hence their aggregates must be replaced accordingly, although probably not in a FIFO fashion. As explained in [14], aggregation over a time-based sliding window is weak non-monotonic without using negative tuples, since a scalar aggregate expires when substituted by a new value for the respective group at the materialized table. We argue that this explanation overlooks the extreme case of a group with no fresh contributing tuples, which eventually all get extinguished from window state. As a plausible remedy to handle "vanishing" groups, we systematically keep the current count c_i of tuples for each group identifier g_i, and eliminate any group g_i as soon as it contains no tuples anymore. With a similar reasoning, we trivially assert that aggregation over other logical (tumbling, landmark) windows is also weak non-monotonic.

Maintaining aggregates over physical windows is more troublesome, since aggregates may be recomputed for each arriving tuple. For a count-based window covering the latest n tuples, a newly inserted item may belong to a diverse group

Algorithm 1 Windowed Aggregation

Procedure *UpdateAggregatesExp* (window $\mathcal{W}$, expiring tuple s, $\mathcal{L}$, $\mathcal{A}_m$, f)
Input: materialized result $\mathcal{M}$ with scalar aggregates per group;
Input: hash index $\mathcal{H}$ for current state of window $\mathcal{W}$;
$g \leftarrow \langle s.\mathcal{A}_1, \ldots, s.\mathcal{A}_k \rangle, A_i \in \mathcal{L}, i = 1..k$; *//Identify group from attribute values*
$v \leftarrow s.\mathcal{A}_m$; *//Value at the aggregated attribute*
Locate record $\langle g, a, c \rangle$ corresponding to g in materialized table $\mathcal{M}$;
$c \leftarrow c - 1$; *//Decrement count of tuples for group g*
if $c = 0$ **then**
  Delete record $\langle g, a, c \rangle$ from $\mathcal{M}$; *//g just became a vanished group*
  return;
end if *//Next, update aggregate value of a depending on function specifics*
if $f \in \{$`SUM`, `AVG`$\}$ **then**
  $a \leftarrow a - v$;
else if f is `COUNT` **then**
  $a \leftarrow a - 1$;
else if $f \in \{$`MIN`, `MAX`$\}$ **and** $v = a$ **then**
  Scan $\mathcal{H}[g]$ to find item r with suitable f value at attribute $\mathcal{A}_m$;
  $a \leftarrow r.\mathcal{A}_m$;
end if
End Procedure

compared to the oldest item being evicted from state. Hence, possibly two scalar aggregate values get affected by each window movement. Still, negative tuples are not required, since aggregate values in the materialized result are updated according to current window state and potential expiration of "vanishing" groups g_i can be controlled by checking their tuple counters c_i. Hence, aggregation for count-based windows is considered weak non-monotonic and can eliminate stale results eagerly. Trivially, the situation is similar for partitioned windows as well.

Computing aggregates poses some additional challenges. Although `SUM`, `COUNT`, `MIN`, `MAX` are distributive and `AVG` is algebraic [15], they require different handling. Indeed, `SUM` can be computed incrementally in constant time, by simply subtracting values from expired items and adding newly arrived ones to the scalar aggregate. Similarly, for an expiring item, the `COUNT` of its respective group g_i gets decremented, while each fresh tuple increments the count of a group g_j to which it gets assigned. Accordingly, `AVG` can be derived by dividing partial sums and counts for each group, hence in terms of tuple processing is identical to `SUM`. We call `SUM`, `COUNT`, and `AVG` *progressive* functions over windows (termed "subtractable" in [6]), since they can maintain aggregated results by only examining "delta" changes in the window state.

But maintaining `MIN` and `MAX` over a set of values could require a sequential scan of the respective group, thus incurring linear time with the tuple count of that group. We term both functions as *retrospective*, because for any window variant, if the local min/max value in group g_i has just expired, tuples in g_i should be probed one by one in order to identify the new aggregate; in case of a single group, the entire window state must be inspected (Fig. 4b). But this costly

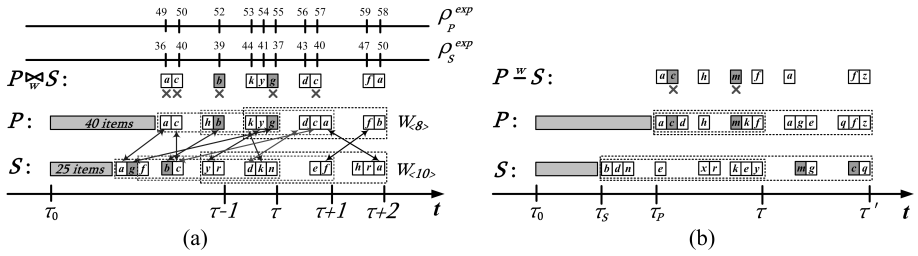

Fig. 5. Binary windowed operations over streams P and S. (a) Join over count-based windows, considering the latest 8 items from P and 10 items from S. Join result g is emitted after b, but expires earlier than b. (b) Result c of the difference between landmark windows expires after invalidation of a later item m, once c is seen on S.

sequential scan is only necessary when the tuple offering the min/max value gets discarded from window state. Unless streaming items have non-increasing values at their aggregated attribute with function `MAX` (and non-decreasing values for `MIN`), successive expirations and thus exhaustive scans are less frequent. Luckily, an incoming item that does not qualify as a min/max (compared to the current scalar aggregate) is simply ignored and no state probing occurs. Therefore, under a random distribution of values at the aggregated attribute $\mathcal{A}_m$, evaluation of functions `MIN` and `MAX` expectedly has amortized constant cost. As a proof of concept, Algorithm 1 outlines a simple method for recomputing aggregates due to tuple expirations; handling updates to aggregates in case of tuple arrivals works similarly. Of course, more optimized techniques could be devised, like the one proposed in [6] for the special case of sliding window aggregates. Similar optimizations are outside the scope of this paper.

Windowed Join. At each $\tau \in \mathbb{T}$, the *windowed join* between two streams returns concatenated pairs of tuples taken from either window state that match according to join criteria $\mathcal{J}$ on certain corresponding attributes:

$$S_1(\tau) \overset{\bowtie}{_W} S_2(\tau) = \mathcal{W}_1(S_1(\tau)) \ \bowtie \ \mathcal{W}_2(S_2(\tau)) = \{\langle s_1 \circ s_2, \tau_m \rangle : \\ s_1 \in \mathcal{W}_1(S_1(\tau)),\ s_2 \in \mathcal{W}_2(S_2(\tau)) \wedge \ \mathcal{J}(s_1, s_2) \wedge \ \tau_m = f(s_1.\mathcal{A}_\tau, s_2.\mathcal{A}_\tau)\}.$$

This symmetric binary operator holds even for diverse window specifications in either stream and can be easily generalized for multi-way joins. Without loss of generality, we consider equi-joins with conjunctive conditions as the most widely used type of joins. In that case, two separate hash tables $\mathcal{H}_1, \mathcal{H}_2$ can be maintained for each stream using values at join attribute(s) as keys, so as to facilitate checks for matching tuples. Then, evaluation is performed as a *symmetric hash join*: each incoming tuple from either stream is hashed against the opposite hash table to identify potential matches with windowed tuples of the other stream (pointed to with arrows in Fig. 5a). If matching tuples are found, the resulting concatenated element gets a new timestamp value τ_m assigned by a function f (e.g., `min`) on the original timestamps or a system-generated time indication.

But our main concern is expiration of windowed tuples, as it affects consistency of join output. The simple case is a join over two landmark windows, which is apparently monotonic since no expirations occur and the output is append-only; once a new tuple s_1 matches an existing item s_2 from the other window, generated item $s_1 \circ s_2$ will remain valid forever.

For the remaining window types, the following rule is applicable: *a joined item should expire as soon as one of its constituent tuples gets discarded from its respective window state*. Note that a fresh result may be deleted soon after its production and before other previously issued results, if it represents a matching with an old stream tuple that expires shortly (e.g., result g in Fig. 5a). This rule poses no problem when expiration is signaled with explicit time indications from system-wide clocks that accomplish a common reference for all tuples. This is the case when logical (time-based sliding or tumbling) windows are specified for joins: the expiration timestamp of result $s_1 \circ s_2$ is simply the earliest (i.e., minimum) expiry time indication at original tuples s_1, s_2. Since joined tuples cannot be maintained in a FIFO fashion, but have known expirations, time-based window joins are weak non-monotonic. Thus, validity of results is determined beforehand and the output always remains consistent, avoiding probing of window states.

Nonetheless, for joins over physical windows, expiration order of tuples is determined separately for each window, according to diverse stream arrival rates for count-based frames, whereas for partitioned constructs it also depends on varying frequencies of group identifiers (Section 3.3). Consequently, we cannot compare expiration orders ρ_1^{exp} and ρ_2^{exp} for any matching tuples s_1 and s_2, so as to decide expiration of their joined result. To overcome this peculiarity, we advocate that each result should carry both expiration orders and we index resulting tuples in two separate catalogues. As becomes evident from the example in Fig. 5a, the sequence of expirations in each catalogue is not FIFO. However, if we maintain each catalogue sorted by expiration order (key) and each entry points to the respective joined result, then we can eliminate stale answers efficiently.

This removal can be carried out lazily, by periodically discarding obsolete results with a batch "garbage collection" process. After polling a window to learn its earliest expiration order ρ_{min}^{exp}, it is safe to invalidate multiple join results by simply following the links from entries up to key ρ^{exp} at the respective catalogue. The same process is repeated for the other window. Yet, we opt for an eager approach to maintain a consistent output, by eliminating results immediately after window expirations. Once a constituent tuple s is discarded from window, through its ρ^{exp} we can locate from the catalogue and purge all results to which s has ever contributed. This is certainly more expensive than the lazy policy, due to many isolated accesses to operator state as opposed to a massive removal. But from a semantics point of view, the eager strategy seems more adequate since outdated answers are never retained.

4.3 Inflexible Operators

There are cases that an operator could possibly ignore update patterns for tuples in its associated window. Results could be cancelled irregularly and their validity

is hardly controlled with expiration timestamps. Difference over two windows, also called *negation* in [14], is a typical example of such an *inflexible* operator.

Windowed Difference. As for union, we assume multiset semantics, by properly adjusting tuple multiplicities. Then, taking the *difference* between window states of two streams with compatible schemata yields at every $\tau \in \mathbb{T}$:

$$S_1(\tau) \stackrel{w}{-} S_2(\tau) = \mathcal{W}_1(S_1(\tau)) - \mathcal{W}_2(S_2(\tau)) = \{\langle s, k\rangle : \exists\ k_1, k_2 \in \mathbb{N},\\ \langle s, k_1\rangle \in \mathcal{W}_1(S_1(\tau)) \wedge \langle s, k_2\rangle \in \mathcal{W}_2(S_2(\tau)) \wedge k = \mathsf{max}(0, k_1 - k_2) \wedge k \neq 0\}.$$

Each qualifying tuple s from stream S_1 retains its original timestamp $s.\mathcal{A}_\tau$ in the materialized result. Expiration times assigned to the matching tuples play a rather secondary role in result maintenance. Multiplicity k of result tuple s is reduced whenever a similar s expires from state $\mathcal{W}_1$. If s expires from state $\mathcal{W}_2$ and matches with an existing s from $\mathcal{W}_1$, then k of the resulting s is increased; otherwise, that expiration causes no trouble.

But difference between windows $\mathcal{W}_1$ and $\mathcal{W}_2$ demonstrates a *non-monotonic* pattern, because tuples may be removed unpredictably from the answer. Indeed, as soon as a new item s arrives in $\mathcal{W}_1$ and is not present in $\mathcal{W}_2$, it is appended to the result, i.e., multiplicity k of resulting tuple s is incremented. If a similar item s appears in $\mathcal{W}_2$ and matches an existing s anywhere in $\mathcal{W}_1$, multiplicity k of the latter item gets decremented; in case $k = 0$, item s is removed from the answer. But the series of such removals do not generally coincide with the insertion order of tuples in the result. Difference is non-monotonic even for landmark windows, because expiration of output tuples can occur at arbitrary times, whenever a match is found between the growing contents of input states (Fig. 5b).

This intransigent behavior of difference can be attributed to its intrinsic relational semantics, which makes negative tuples indispensable for explicit result deletions in all its windowed versions:

Proposition 8. *Difference between any window variants is non-monotonic.*

4.4 Applying Diverse Windows over Binary Operators

Up to this point, we assumed that both inputs to binary operators (union, join, difference) were similar window types, although they may differ in their exact signatures, i.e., parametric properties like temporal extent, progression step etc. However, it may seldom occur that diverse window types are specified. Suppose a monitoring application receiving data from a stream P with humidity readings and another stream S with temperatures at the same stations. If we want to calculate a heat index ("how hot it feels") from such meteorological variables, we can submit the following continuous query:

```
[Q3]: SELECT P.stationID, MAX(P.humidity), AVG(S.temperature)
      FROM P [UNBOUNDED START AT '07:00:00'],
           S [RANGE 60 MINUTES SLIDE 1 MINUTE]
      WHERE P.stationID = S.stationID AND S.Temperature >= 30
      GROUP BY P.stationID
```

Query Q3 is a join between a landmark window (for taking peak humidity measurements after sunrise) and a time-based sliding one (high temperatures over past hour), followed by aggregation to get the values needed for the heat index at each station. Obviously, Q3 is weak non-monotonic since join involves a sliding window that influences the result more than its monotonic landmark counterpart. Although many combinations of windows with a variety of signatures may arise in actual query specifications, it can be easily verified that:

Proposition 9. *The more strongly non-monotonic a window is, the more it dominates the behavior of the binary operator.*

Such monotonic-related patterns concerning windowed operators can definitely improve optimization of execution plans for continuous queries, in addition to algebraic rules involving windows, discussed next.

5 Syntactic Equivalences involving Windows

In this section, we turn our attention on syntactic equivalences in query expressions that specify window frames. Although this is not an exhaustive investigation of such properties, it comes naturally as a consequence of windowing semantics and emphasizes their usefulness in query rewriting.

Interestingly, when intertwined with other operators, logical (time-based sliding, tumbling and landmark) windows are more flexible than physical ones (i.e., count-based, partitioned). First, it is evident that logical windows specified on the basis of totally ordered timestamps of streaming items are commutative with all *neutral* operators. Results would be identical either the window is first being applied on the stream or on the output of the relational operator, since ordering of tuples is preserved in both cases thanks to timestamping. More specifically:

- *Projection:* $\pi_{\mathcal{L}}\ (\mathcal{W}_P(S(\tau))) = \mathcal{W}_P(\pi_{\mathcal{L}}(S(\tau)))$
- *Selection:* $\sigma_{\mathcal{F}}\ (\mathcal{W}_P(S(\tau))) = \mathcal{W}_P(\sigma_{\mathcal{F}}(S(\tau)))$
- *Union:* $S_1(\tau) \bigcup\limits_{w} S_2(\tau) = \mathcal{W}_P(S_1(\tau) \bigcup S_2(\tau)) = \mathcal{W}_P(S_1(\tau)) \bigcup \mathcal{W}_P(S_2(\tau))$

Note that the rule for union holds if identical window signatures are specified over both streams.

In contrast, physical windows commute with projections only, since some attributes are merely dropped. But the state of a count-based window may contain different items if selection has been formerly applied against the entire stream. Instead of getting the n most recent items, the result would be again a set of n tuples, but some of them might not be that fresh, depending on selection predicates. As for union, if each count-based window is applied separately, n tuples will be returned from each stream, leading to a wrong total of up to $2n$ items for their temporary union. Note that unified results must be merged and synchronized by their arrival, which clearly cannot be preserved due to diverse orderings and possibly fluctuating rates in contributing streams. The same arguments also hold for partitioned windows.

With regard to *interfering* and *inflexible* operators, windows are generally not commutative, since these operators maintain state and are not simple filters over streams. The only exception is duplicate elimination which commutes solely with logical windows following a reasoning similar to selection:

– *Duplicate elimination:* $\delta\ (\mathcal{W}_P(S(\tau))) = \mathcal{W}_P(\delta(S(\tau)))$

Still, windowed analogues of *joins* have some interesting properties:

– *Commutative:* $S_1(\tau) \overset{\bowtie}{}_W S_2(\tau) = S_2(\tau) \overset{\bowtie}{}_W S_1(\tau)$
– *Associative:* $(S_1(\tau) \overset{\bowtie}{}_W S_2(\tau)) \overset{\bowtie}{}_W S_3(\tau) = S_1(\tau) \overset{\bowtie}{}_W (S_2(\tau) \overset{\bowtie}{}_W S_3(\tau))$
– *Distributive over selection:* $\sigma_{\mathcal{F}}(S_1(\tau) \overset{\bowtie}{}_W S_2(\tau)) = \sigma_{\mathcal{F}}(S_1(\tau)) \overset{\bowtie}{}_W \sigma_{\mathcal{F}}(S_2(\tau))$ and *over projection:* $\pi_{\mathcal{L}}(S_1(\tau) \overset{\bowtie}{}_W S_2(\tau)) = \pi_{\mathcal{L}}(\pi_{\mathcal{L}_1}(S_1(\tau)) \overset{\bowtie}{}_W \pi_{\mathcal{L}_2}(S_2(\tau)))$.

Note that distributive property for selection holds for logical windows only. Attribute lists $\mathcal{L}_1$ and $\mathcal{L}_2$ in separate projections over each stream must include attributes from $\mathcal{L}$ and any attributes involved in join conditions $\mathcal{J}$. Similar properties are also applicable to *union* over logical windows only.

Certainly, the purpose of such rules is optimization of logical query plans. As in this paper we only deal with autonomous windowed operators, we refrain from further discussion over rules for pushing down selections, subquery flattening etc. However, we presume that the labelling rules for query plans specified in [14] with respect to update patterns of time-based sliding windows, also hold for other windowing constructs. This topic certainly requires more investigation and is left for future research.

6 Empirical Validation

In this section, we first describe implementation specifics of our semantics framework and then report comprehensive results from its empirical validation against a real dataset.

6.1 Operator Implementation

Windowed adaptations of relational operators were implemented for all variants described in Section 3, in order to verify feasibility of our semantics foundations with particular emphasis on update patterns. Admittedly, our approach stems from a theoretical background and attempts to validate its correctness with respect to consistency of operator results. Thus, processing algorithms lack proper optimization, and certainly cannot be compared to full-fledged stream prototypes [2, 7, 11]. We stress that our main concern is to take advantage of monotonic-related patterns and maintain consistently the state of *autonomous operators*, rather than execute physical query plans involving a series of interconnected operators. As ours is not a complete processing mechanism, we do not undertake a performance study for composite continuous queries, e.g., according to the Linear Road Benchmark [5], emphasizing on the behavior of individual operators instead.

Operators were implemented as separate classes in C++ and abstracted as iterators consuming tuples from their input queue and feeding with results their output queue. Queues can serve as a means for inter-operator connectivity, but also for incremental availability of ordered results.

A quite valuable feature of our framework refers to the practically negligible cost of maintaining autonomous window states, thanks to their intrinsic update patterns. Each such construct is implemented as a virtual queue over the input stream, so no tuples need be copied or deleted when window bounds change. Even partitioned windows, despite that their state may involve detached tuples and not cohesive stream chunks (Fig. 3b), they really incur very low overhead because each incoming item affects just the head and tail of a single partition. However, operator results are materialized (e.g., for aggregates, distinct values or joined tuples), in order to keep track of their validity by eagerly expunging stale items. Finally, hash tables were maintained for evolving window states, in order to speed up retrieval and update efficiency for operators.

6.2 Experimental Setup

Experiments were performed against a real dataset that contains traces of network traffic for 782 281 wide-area TCP connections between the Lawrence Berkeley Laboratory (LBL) and the rest of the world. This dataset is publicly available from `http://ita.ee.lbl.gov/html/contrib/LBL-CONN-7.html` and was also utilized in [14]. For testing binary operators, we break up the initial file into two separate ones (`TraceA` and `TraceB`) with identical attributes. In all cases, we consider streams with the following schema:

```
NetTrace (timestamp, duration, protocol, payload-sent,
          payload-received, localIP, remoteIP, status, flags)
```

where each sequential tuple represents a session. We ignore original time indications, since we utilize each dataset as a stream source at a parameterized *arrival rate* of $\lambda = \{10K, 20K, 50K, 100K\}$ tuples/sec. If logical windows are applied, tuples arriving concurrently at the same "wave" must receive identical timestamps τ; in case of physical windows, stream items are assigned successive admission orders (Section 3). We also implemented an auxiliary *scan* operator, which simulates a given rate λ by reading a suitable chunk of input at each execution cycle τ and then feeds the specified window. Hence, input tuples are always accepted in timestamp order, so stream imperfections such as delayed or out-of-order items cannot occur [27].

As a rule in data stream processing, we adhere to online in-memory computation, excluding the use of hard disk for performance reasons. Experiments with diverse parameter settings were simulated on an Intel Core 2 Duo 3GHz CPU running GNU/Linux with 2GB of main memory. Unless otherwise specified, results are averages of actual measurements at each execution cycle over complete (i.e., not "half-filled") windows, after an initialization phase in which frames get populated up to their specified capacity.

6.3 Experimental Results

Since neutral operators (i.e., projection, selection, and union) act as simple filters over windows, their results get updated in FIFO exactly as prescribed by the respective frame. As they do not impose any particular overhead in terms of monotonic-related patterns, we do not report performance results for them. We also exclude negation from our experiments, since that operator has an arbitrarily disordering effect on emitted results and does not tolerate any meaningful update patterns.

We conducted a series of experiments especially concerning *interfering* operators, due to their active interplay with window patterns and their resource requirements for maintaining consistent states. We have tested implementations of three types of continuous queries, each executed in isolation and involving a single operator (respectively for duplicate elimination, aggregation and equijoin), but coupled with various window signatures. Concerning aggregation, we have tested the behavior of `SUM` and `MAX`, to strike the difference between progressive and retrospective functions. These simple queries have clear semantics and could be expressed as follows in CQL:

```
[Q4]: SELECT DISTINCT protocol, localIP, remoteIP
      FROM NetTrace [Window signature];

[Q5]: SELECT localIP, AGGR(duration)
      FROM NetTrace [Window signature]
      GROUP BY localIP;

[Q6]: SELECT *
      FROM TraceA [Window signature], TraceB [Window signature]
      WHERE TraceA.localIP = TraceB.remoteIP;
```

Placeholder `AGGR` is used to denote either `SUM` or `MAX`, while the exact `[Window signature]` is determined by parameters set for each experiment. Next, we comment on update patterns observed in these queries specifically for each window variant. Reported measurements exclude execution costs for window state maintenance, since they are similar under identical signatures.

Time-based Sliding and Tumbling Windows. Such frames are advantageous with respect to state maintenance, since all expiring items from windows can be removed at once, thanks to the total order of their timestamp values. Figure 6 illustrates the execution cost of each operator under varying slide steps (every β timestamps) for a fixed window extent $\omega = 10$ timestamps. All operators have linearly escalating processing times for increasing sliding steps. This is quite expected under a steady arrival rate of 10K tuples/sec, since an equal number of items expire at each window movement. Join is far more expensive, due mainly to the cost of potentially producing multiple matching tuples for each incoming one. However, outdated results can be all purged together, since they are appropriately sorted by their expiration timestamps. In contrast, aggregation has an almost negligible cost. Indeed, upon expiration of a `MAX` value, even though search for its replacement must access the entire window state, it runs only once every

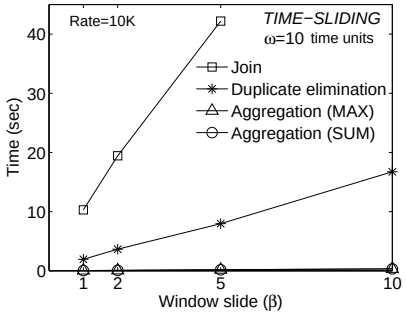

Fig. 6.

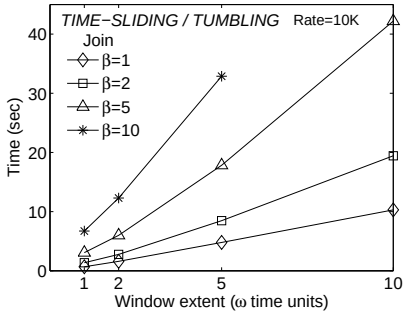

Fig. 7.

β units. Similarly, for `SUM` only "delta" changes in window state need be considered, i.e., fresh tuples appended to its upper bound and remote items expired from the lower one. Eliminating duplicates is more costly, since any distinct value expiring from window incurs a separate state scan for its replacement. But, as this search is carried out "backwards", in most cases a younger substitute could be found very soon.

Greater temporal extents also impose an almost linear increase in operator cost. As depicted in Fig. 7, the cost of joins clearly depends on window size, as more opportunities for matching tuples may occur for larger frames. Recall that if $\beta \geq \omega$, the window frame is essentially a tumbling one, but it can be maintained as if it were a sliding one (Section 3.2.2).

Count-based Windows. The main difference of these constructs from their time-based counterparts is that expirations are done on a per tuple basis and not in batch mode. Normally, it should be expected that expensive operators like join could have a rapidly escalating cost for larger windows. As plotted in Fig. 8, this is certainly inefficient in case of explicit removals with negative tuples (*NT*), due to the overwhelming cost of identifying anew matching tuples as negatives and purging them separately. Even though we index results on their values on the join key, obsolete ones are only removed from the respective list after a sequential scan of its entries. But if results are indexed according to expiration orders of their constituent tuples (Section 3.3), then purging of stale joined items ordered by expiration (*ORD*) can have a significant benefit of more than two orders of magnitude. This observation indicates that even a simple index can prove valuable to smooth maintenance of consistent operator states. Thus, all subsequent graphs refer to this expiration policy, without making use of negative tuples.

With respect to operator costs, Fig. 9 shows a slightly different pattern than time-based windows. For count-based constructs, joins are again more costly since they could generate multiple output items. Duplicate elimination has a moderate cost, while maintenance of `SUM` aggregates can be carried out efficiently by examining only two tuples for each pair of insertions and deletions of window

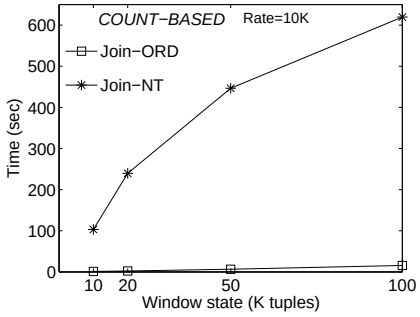

Fig. 8.

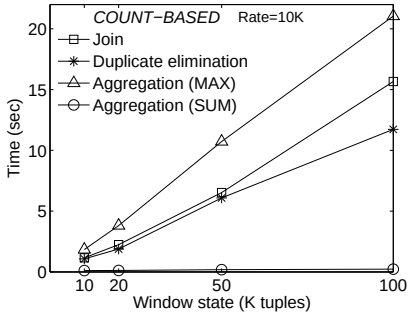

Fig. 9.

state. For these windowed operators, only a local arrangement suffices to refresh results by replacing a tuple or adjusting a group in current state. But handling of `MAX` aggregates is the most expensive case for count-based windows. The reason is that expiration of a local maximum always incurs a complete sequential scan of items in the respective group. If this search for replacements is frequent for expiring tuples, then execution cost grows linearly with the window size. This is clearly an inherent limitation as opposed to the flexibility of these retrospective functions with time-based sliding frames. Figure 10 illustrates execution time for eliminating duplicates under diverse stream arrival rates. Clearly, the cost increases almost linearly in direct analogy with the size of window states. Similar measurements were obtained for other operators as well and thus omitted for brevity. Finally, we mention that duplicate elimination has significant initialization time (figures not reported in the graphs) when windows are not yet filled and distinct values are collected, but almost constant time at each renewal of full-capacity window states.

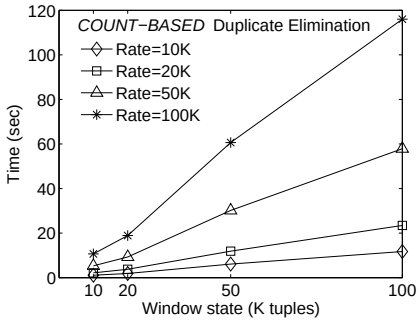

Fig. 10.

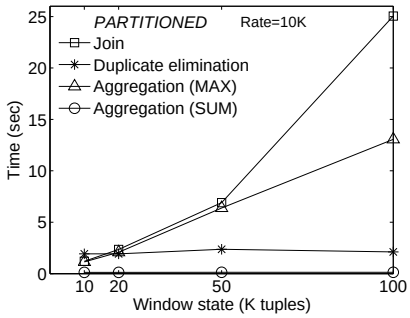

Fig. 11.

Partitioned Windows. We define partitioned windows on attribute `protocol`, but since some resulting partitions carry very little data, we maintain only the 5 more populated ones in the stream. In Fig. 11, state size refers to the total amount of tuples from all partitions. Interestingly, the cost for eliminating duplicates is almost stable, no matter the window size. This occurs because the partitioning attribute is also used for determining distinct values (Query Q4), hence searching for substitutes of expired values always takes place in the same partition, which is only a subset of the overall window state. This feature also explains why the cost for identifying `MAX` values is reduced in comparison to joins and almost halved compared with duplicate elimination over count-based windows. Note that absolute costs differ with regard to operators over count-based frames of equal size (Fig. 9), as respective window state(s) may include the same number of items but actually contain different ones, because partitions control their contents on their own (Section 3.3.2).

Landmark Windows. Such primitives are distinctively different, thanks to their monotonic behavior. Figure 12 plots the cumulative execution time along certain snapshots of window state. Because no items ever expire from state, landmark windows are very effective when combined with duplicate elimination or aggregation. Their processing cost is almost negligible even for large numbers of state tuples, as there is no need to check with any prior stream items. Only operator results (i.e., distinct or aggregate values) may be affected, if a new tuple is not a duplicate or changes the existing aggregate for its group. But the situation differs for joins, displaying an aggravating effect on execution cost. As window states expand and more items get gradually accumulated in either window state, many more opportunities for matching tuples could exist depending on the selectivity of join conditions.

In addition, operator state maintenance incurs extra space overhead. Naturally, the landmark window itself shows a linearly increasing memory footprint as more tuples get included into its state, which grows steadily with new stream arrivals. Not surprisingly, with window types other than landmark ones, we ob-

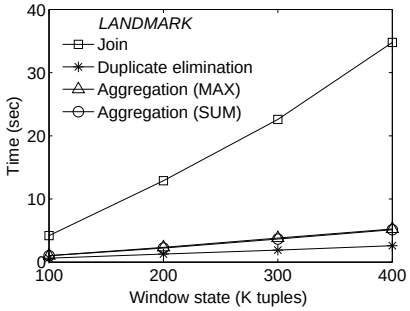

Fig. 12.

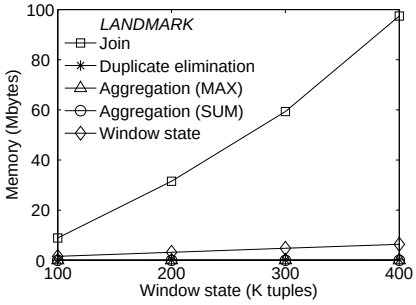

Fig. 13.

served that memory nearly stabilizes to a limited amount that depends only on window extents, assuming a constant input rate (results are trivial and thus omitted). But for the operators themselves, memory cost for result maintenance depends on their type. As depicted in Fig. 13, join is the most demanding for landmark windows, because it requires maintenance for increasing numbers of matching tuples. In contrast, duplicates get discarded immediately, so space requirements of distinct values are almost negligible. The situation is similar with aggregates, with no variation among the specified functions, since the number of groups is identical for both.

Discussion. Empirical results clearly indicate that it is the combination of windows to relational operators that really matters in query evaluation. Apparently, the actual contents of window states vary according to the inherent semantics of each particular variant. In fact, the actual state depends on whether or how window bounds change. Such state alterations may cause serious problems at already emitted or retained results and thus require immediate action. Of course, maintenance of consistent output comes at a possibly significant cost, especially for certain crucial operations like joins, duplicate elimination and some aggregate functions. But we believe that it is worth the price to always guarantee safe results over such volatile datasets. Last, but not least, with this empirical validation we also affirmed that our interpretation of window and operator semantics perform exactly as expected and the relevant structures always get updated in the prescribed manner under diverse stream rates and window signatures.

7 Related Work

Over the past few years, there has been a surge of research interest on managing data streams, offering a variety of sophisticated algorithms and innovative evaluation strategies. Next we review topics particularly related to window semantics and continuous queries. The first notion of continuous queries over append-only databases appeared in Tapestry [29] as a means to provide timely responses by utilizing periodic query execution and identifying several rewriting rules for incremental evaluation. In [8], this approach was enriched such that continuous semantics could deal with more involved cases than insertions, i.e., allowing deletions or modifications in the database. In [14], a more precise definition is given, taking into account both streams and updatable relations; but updates to already received items are never allowed in order to guarantee consistency for previously emitted answers.

In practice, several renowned system prototypes have been implemented, each involving varying forms of windowing constructs, considered indispensable for real-time processing [27]. Of the most prominent stream engines, AURORA [1] and its distributed version Borealis [2] are data flow oriented systems, which utilize sliding and tumbling windows for aggregation, join and sorting of output results. STREAM [7] offers a Continuous Query Language (CQL) equipped with time-based and tuple-based sliding windows with resource sharing capabilities [6], as well as partitioned windows adhering to the SQL:1999 standard [4].

As for TelegraphCQ [11], only time-based sliding windows are available in its StreaQuel language, but support for hopping (i.e., tumbling) frames could also be achieved. No windowing constructs are explicitly specified in Gigascope [17], but their semantics are indirectly expressed as constraints involving monotonically increasing timestamps of input streams. As an alternative to windows, punctuations were introduced in [30, 31], by suitably embedding special signs in the stream that denote the end of a subset of data. Following this policy, Gigascope regularly generates punctuations (*"heartbeats"*) in order to unblock operators like aggregation in query plans. Recently, a stream engine built on top of a column-oriented DBMS was presented in [20], proposing transitory storage of incoming tuples in suitable system tables called baskets. Items are then propagated to operators after filtering by means of predicate windows, which apply simple selection conditions on the basket data, even irrespective of their timestamp order. Last, but not least, window functions are also important for processing XML streams and a related extension to XQuery has been suggested in [9] offering landmark, sliding and tumbling windows.

Taking advantage of such solid research foundations, several commercial products have begun emerging in the market, offering powerful stream processing capabilities. StreamBase platform [28] builds on the experience of Aurora and Borealis prototypes and provides real-time event processing with a mature StreamSQL language for specifying continuous queries. Their model is tuple-driven, so query evaluation is performed according to the arrival order of stream items. In contrast, Oracle's Complex Event Processor [23] uses an extension of CQL that adheres to a time-driven scheme, hence window states evolve according to the timestamp values of incoming tuples. Coral8 [3] in its Continuous Computation Language (CCL) employs stream manipulation according to both tuple- and time-driven approaches and utilizes windowing constructs that can be shared by multiple queries.

Towards a streaming SQL standard, a hybrid data model was recently proposed in [16], which attempts to bridge the gap between tuple-driven and time-driven semantics. The underlying concept is that evaluation emanates from the arrival of a batch of tuples, that can be considered either as items of identical timestamp value or distinctly ordered tuples. A novel stream-to-stream operator SPREAD is used to provide fine-grained control over ordering relationships among tuples, such that the conflicting demands of simultaneity and ordering can be both captured.

Regarding processing techniques using windows, the interesting idea of *negative tuples* [13] has been suggested for evaluating sliding window queries, as a means of cancelling results that are no longer valid. This approach entails drastically revised semantics and implementation changes for most operators, since both regular (positive) and expiration (negative) tuples must flow through query execution plans. Several optimization techniques were applied for reducing the overhead of doubling the amount of tuples processed and hence to avoid output delays. Taking into account the theoretical framework in [21], a detailed examination focusing strictly on windowed aggregates was proposed in [19], cov-

ering sliding, landmark and partitioned frames. Under this interpretation, an additional attribute is attached to the grouping list used for aggregation; then, windowed aggregation reduces to a simple relational one. One of the goals is to deal with disorder in incoming stream elements, hence any attribute with a totally ordered domain (and not just timestamps or sequence numbers) can be declared as the windowing attribute. Most recently, a temporal foundation for a stream algebra is attempted in [18], which distinguishes between logical and physical operator levels. Transformation rules are provided between a logical level that refers to query specification and a physical level that includes actual implementation specifics for operators. In terms of windows, though, only time-based sliding variants are supported.

To the best of our knowledge, update patterns for sliding window operators have been analyzed primarily in [14], aiming to exploit such properties in operator evaluation at physical execution plans. Apart from introducing a suitable classification of operators with respect to monotonicity, an update-pattern-aware query processor is also built, capitalizing on the usage of expiration times and offering optimization rules for avoiding negative tuples. This important work is perhaps the closest in spirit with our own, but their objective is entirely concentrated on time-based sliding windows. A recent approach in [10] also exploits these update patterns so as to configure and fine-tune access to the data, disassociating stream storage management from the actual processing mechanism. Besides, in [26] expiration times are deemed significant even for relational tables, as a means of signaling the lifetime of data therein. Their proposed algebraic extensions distinguish operators with respect to monotonicity, but the model has been devised particularly for synchronizing loosely coupled information systems, without taking into account the real-time characteristics of streaming data. In our work, we develop a generalization of monotonic-related patterns in windows and carefully investigate their impact on typical query operators in more detail than originally discussed in [25]. Enhancing previous studies, we analyze all frequently used window variants over data streams according to rigorous semantics for a rich set of window types [24]. We believe that such sound algebraic formulation of windowed operations offers a deep insight on their intrinsic features and can assist to efficient evaluation strategies particularly geared towards consistency of query results.

8 Conclusions

In this paper, we exhibit the significance of windows on continuous query evaluation, by deeply understanding interesting update patterns intrinsically tied to their semantics. We developed a foundation with clear semantics and analyzed diverse forms of windowing primitives already in widespread use in stream processing engines. In addition, we presented adaptations of principal relational operators that can express a wide range of user requests.

Although most window frames do not generally refresh their contents in a truly monotonic fashion, we showed that opportunities still exist for sig-

nificant savings in their efficient maintenance, mitigating the burden of non-monotonicity. We pointed out that operator output must always be kept updated, not only with new results, but also removing stale items that have expired from the respective windows. Further, our careful algebraic formulation for the windowed operators is of particular importance when checking for syntactic equivalences. We complemented our analysis with a comprehensive validation that gives sufficient evidence of its significance, to the benefit of advanced query evaluation. Overall, we consider that a rich set of windows play a vital role in stream management and that they should be given first-class citizenship in a long-expected stream algebra.

Further improvement is possible with respect to shared evaluation and query optimization in the presence of overlapping window states. In line with efforts towards foundation of a stream algebra, we also plan to examine properties concerning multiple windows and query rewriting rules in composite execution plans. Finally, efficiently indexing massive amounts of volatile stream items is also a challenging topic, in order to provide the means for superior state maintenance of valid query results.

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. D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, January 2005.
3. Aleri Inc. Continuous Computation Language Reference (2008) available at `http://www.coral8.com/WebHelp/coral8_documentation.htm`
4. A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2):121-142, 2006.
5. A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB*, pp. 480-491, September 2004.
6. A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*, pp. 336-347, September 2004.
7. 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.
8. D. Barbarà. The Characterization of Continuous Queries. *International Journal of Cooperative Information Systems*, 8(4): 295-323, 1999.
9. I. Botan, P.M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *VLDB*, pp. 75-86, September 2007.
10. I. Botan, G. Alonso, P.M. Fischer, D. Kossmann, and N. Tatbul. Flexible and Scalable Storage Management for Data-intensive Stream Processing. In *EDBT*, pp. 934-945, March 2009.
11. 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*, January 2003.

12. T. Ghanem, W. Aref, and A. Elmagarmid. Exploiting Predicate-window Semantics over Data Streams. *ACM SIGMOD Record*: 35(1): 3-8, 2006.
13. T. Ghanem, M. Hammad, M. Mokbel, W. Aref, and A. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(1): 57-72, January 2007.
14. L. Golab and M. Tamer Özsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *ACM SIGMOD*, pp. 658-669, June 2005.
15. J. Gray, S. Chaudhuri, A. Boswarth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1), 29-53, 1997.
16. N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a Streaming SQL Standard. In *VLDB*, pp. 1379-1390, August 2008.
17. T. Johnson, S. Muthukrishnan, V. Shkapenyuk, O. Spatscheck. A Heartbeat Mechanism and its Application in Gigascope. In *VLDB*, pp. 1079-1088, September 2005.
18. J. Krämer and B. Seeger. Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. *ACM TODS*: 34(1), Article 4, April 2009.
19. J. Li, D. Maier, K. Tufte, V. Papadimos, P. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *ACM SIGMOD*, pp. 311-322, June 2005.
20. E. Liarou, R. Goncalves, and S. Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In *EDBT*, pp. 323-334, March 2009.
21. D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of Data Streams and Operators. In *ICDT*, pp. 37-52, January 2005.
22. J. Melton. *Advanced SQL:1999 - Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, USA, 2002.
23. Oracle Inc. Complex Event Processing in the Real World (2007), available at `http://www.oracle.com/technologies/soa/docs/oracle-complex-event-processing.pdf`
24. K. Patroumpas and T. Sellis. Window Specification over Data Streams. In *ICSNW*, Springer LNCS 4254, pp. 445-464, March 2006.
25. K. Patroumpas and T. Sellis. Window Update Patterns in Stream Operators. In *ADBIS*, Springer LNCS 5739, pp. 118-132, September 2009.
26. A. Schmidt, C.S. Jensen, and S. Šaltenis. Expiration Times for Data Management. In *IEEE ICDE*, p. 36, April 2006.
27. M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 Requirements of Real-Time Stream Processing. *ACM SIGMOD Record*, 34(4):42-47, December 2005.
28. StreamBase Systems. StreamSQL Guide (2009) available at `http://www.streambase.com/developers/docs/sb62/pdf/streamsql.pdf`
29. 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.
30. 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.
31. P. Tucker, D. Maier, T. Sheard, and P. Stephens. Using Punctuation Schemes to Characterize Strategies for Querying over Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(9): 1227-1240, September 2007.