The Extensibility Framework in Microsoft StreamInsight

Mohamed Ali[1], Badrish Chandramouli[2], Jonathan Goldstein[1], Roman Schindlauer[1]

[1]*Microsoft Corporation*

[2]*Microsoft Research*

One Microsoft Way, Redmond WA 98052

{mali, badrishc, jongold, romans}@microsoft.com

Abstract— *Microsoft StreamInsight* (StreamInsight, for brevity) is a platform for developing and deploying streaming applications, which need to run continuous queries over high-data-rate streams of input events. StreamInsight leverages a well-defined temporal stream model and operator algebra, as the underlying basis for processing long-running continuous queries over event streams. This allows StreamInsight to handle imperfections in event delivery and to provide correctness guarantees on the generated output. StreamInsight natively supports a diverse range of off-the-shelf streaming operators. In order to cater to a much broader range of customer scenarios and applications, StreamInsight has recently introduced a new *extensibility infrastructure*. With this infrastructure, StreamInsight enables developers to integrate their domain expertise within the query pipeline in the form of user defined modules (functions, operators, and aggregates).

This paper describes the extensibility framework in StreamInsight; an ongoing effort at Microsoft SQL Server to support the integration of user-defined modules in a stream processing system. More specifically, the paper addresses the extensibility problem from three perspectives: the query writer's perspective, the user defined module writer's perspective, and the system's internal perspective. The paper introduces and addresses a range of new and subtle challenges that arise when we try to add extensibility to a streaming system, in a manner that is easy to use, powerful, and practical. We summarize our experience and provide future directions for supporting stream-oriented workloads in different business domains.

I. INTRODUCTION

Microsoft StreamInsight [11] (StreamInsight, from now on) is a platform for monitoring stream data from multiple sources to extract meaningful patterns, trends, exceptions, and opportunities. StreamInsight analyzes and correlates data incrementally and in-memory while the data is in flight, yielding very low response times and high throughput. StreamInsight is an event stream processing system whose operation and semantics are governed by a temporal extension of relational algebra [3], which logically views a set of events as a time-varying relation. StreamInsight queries consist of a tree of operators with well-defined semantics, as defined by their effect on the time-varying relation. Run-time query composability, query fusing, and operator sharing are some of the key features in the query processor. Further, StreamInsight includes several debugging and supportability tools enable developers and end users to monitor and track events as they are streamed from one operator to another within the query execution pipeline.

The interest in data streaming applications has grown tremendously across various business sectors. For example, streaming engines are used in Web analytics, fraud detection, call center management, RFID monitoring, manufacturing and production line monitoring, smart power meters, financial algorithmic trading, and stock price analysis. A close analysis of a wide range of streaming-oriented workloads reveals the fact that these workloads share common characteristics, and yet, differ from each other in several domain-specific aspects.

The common characteristics include the demand for coping with high event input rates that are usually characterized by imperfections in event delivery (either late events or payload inaccuracies). Further, there is a need for a mechanism to minimize output latency while maintaining correctness guarantees over the output. A streaming engine is an appropriate choice to provide low-latency and high-throughput solutions for these applications. Correctness guarantees with low latency are provided due to the system's ability to (1) output *speculative* [1, 10] results based on potentially incomplete or inaccurate sets of events, (2) *compensate* [1, 22] (or correct) incorrect output as late events and/or more accurate payloads are received at the system's input, and (3) identify which output is *guaranteed* [4, 6] to be correct, i.e., cannot change in the future, based on received (or automatically inserted) guarantees from the event sources.

On the other hand, each business sector has unique demands that are specific to its business logic. Business logic is the outcome of domain expertise in a specific sector over years. It is hard to expect that a single data streaming engine (*out-of-the-box* or *as-it-is* without any specialization) can cover domain expertise in different domains. Thus, for broad applicability, a streaming system is expected to have an extensibility mechanism that can seamlessly integrate domain-specific business logic into the incremental in-memory streaming query processing engine.

As shown in Figure 1, there are three distinct entities that collaborate together to extend a streaming system for a particular application domain:

1- **The user defined module (UDM) writer.** The UDM writer is the domain expert who writes and packages the code that implements a set of domain-specific operations as libraries of UDMs. For example, a financial application may have experts write UDMs

that can detect interesting complex chart patterns [19] in real-time stock feeds.

2- **The query writer.** The query writer invokes a UDM as part of the query logic. A query is expected to have one or more UDMs wired together with standard streaming operators (e.g., filter, project, joins) in the same query pipeline. Note that multiple query writers may leverage the same existing repository of UDMs for accomplishing specific business objectives. The typical query writer does not have a deep understanding of the technical domain-specific details within UDMs, but is an expert at understanding end-user requirements and developing queries to meet these requirements. Continuing the financial example discussed above, the query writer may write a complex query (and the surrounding glue application) that correlates across stock feeds from multiple stock exchanges, performs necessary pre-processing and filtering, applies a UDM to detect a particular chart pattern, and delivers the results as part of a trader's dashboard.

3- **The extensibility framework.** The extensibility framework is an internal system component that connects the UDM writer and the query writer. The UDMs are deployed to the framework and made available to query writers to use as part of their queries. The framework executes the UDM logic on demand based on the query to be executed. Thus, the framework provides convenience, flexibility, and efficiency for both the UDM writer and the query writer.

Microsoft StreamInsight is designed to satisfy the requirements of streaming-oriented workloads. Meanwhile, StreamInsight provides the vehicle to integrate business logic into the execution query pipeline through its extensibility framework. This paper describes the extensibility framework in Microsoft StreamInsight with particular emphasis on the roles of the UDM writer, the query writer, and the system's internals to deliver efficient and extensible solutions to business needs.

A. Design Principles and Contributions

The extensibility framework in Microsoft StreamInsight has been architected to achieve several design principles. These design principles contribute to the flexibility, efficiency and the seamless integration of a wide range of user defined modules into the continuous query processing pipeline. We summarize our design principles as follows:

1) Ease-of-use

This principle favors query writers over UDM writers. A UDM is written once and is used by many queries over time. Hence, UDM details are expected to be hidden from the query writer who invokes the UDM by name and, possibly, passes some initialization parameters if needed.

2) Flexibility and reusability

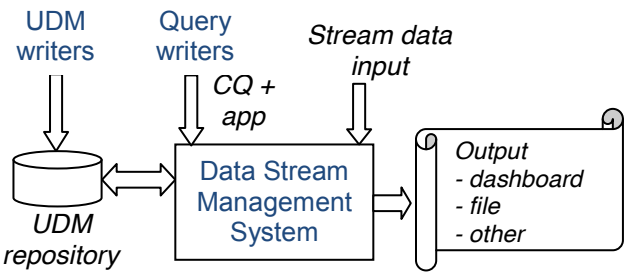

Figure 1: Entities in a streaming solution for a domain.

This principle provides the query writer with the ability to change the event membership in the set of events that are passed to the UDM every time the UDM is invoked. The temporal attributes if input and output events can be altered based on the query semantics. This principle gives the flexibility of re-using the same UDM under different circumstances.

3) Portability and compatibility

This principle favors simple UDM writers or UDM writers who published libraries of UDMs under traditional (non-temporal) database systems. These libraries can be ported to Microsoft StreamInsight with minimal porting effort. Without violating the UDM's view of data as a relational table, StreamInsight handles the temporal aspect of events and handles imperfections in event delivery on behalf of the UDM writer.

4) Powerful control over time-management and efficiency

This principle favors advanced UDM writers who claim responsibility of managing the temporal aspect of events seeking maximum power and efficiency in the UDM. The system provides the UDM writer with the ability to read/generate the input/output timestamps of events. Meanwhile, it provides the query writer with the ability to override the UDM time management and to revert back to default system timestamping policy. Further, it allows advanced UDM writers to increase efficiency in the streaming setting, by providing them with facilities to express incremental computations for the UDM.

5) Breaking optimization boundaries

A UDM stands as optimization boundary in the query pipeline. Because a UDM is a black box to the optimizer, it is hard to reason about optimization opportunities. However, working hand-in-hand with the UDM writer, the UDM writer has the option to provide several properties about the UDM through well-defined interfaces. The optimizer reasons about these properties and shoots for optimization opportunities.

6) Liveliness

Liveliness is a bi-fold aspect: (1) it is the ability to guarantee that output results are accurate and stable (i.e., will not to be retracted) up to a specific point in the application time, and (2) to keep advancing the output time as input time advances. This principle is concerned with generating signals

that advance the output time properly as the system receives signals that indicate advances in the input time.

B. Paper Outline

The remainder of this paper is organized as follows: Section 2 provides some basic background on streams, and introduces our new concept of windowing for the extensibility framework. Then, we cover the details of the extensibility solution, by exploring the problem from three perspectives: the query writer's perspective (Section 3), the user defined module writer's perspective (Section 4), and the system's internal perspective (Section 5). We summarize our experiences and lessons learned in Section 6, and finally conclude the paper in Section 7.

II. Streams, Events, and Windows

A Data Stream Management System (DSMS) [8, 9, 11, 14, 15, 16] is a system that enables applications to issue long-running continuous queries (CQs) that monitor and process streams of data in real time. DSMSs are used for efficient real-time data processing in a broad range of applications. While the core concepts are generalizable to any streaming system, this paper focuses on *Microsoft StreamInsight*, which is based on the CEDR [1, 2] research project.

A. Streams and Events

A physical stream is a potentially unbounded sequence $\{e_1, e_2, \ldots, \}$ of events. An event $e_i = <p, c>$ is a notification from the outside world (e.g., sensor) that consists of: (1) a payload $p = <p_1, \ldots, p_k>$, and (2) a control parameter c that provides metadata about the event. While the exact nature of the control parameter associated with events varies across systems [1, 10, 17], two common notions are: (1) an event generation time, and (2) a duration, which indicates the period of time over which an event can influence output. We capture these by defining $c = <LE, RE>$, where the time interval [LE, RE) specifies the period (or lifetime) over which the event contributes to output. The *left endpoint* (LE) of this interval, also called *start time*, is the application time of event generation, also called the event timestamp. Assuming the event lasts for x time units, the *right endpoint* of an event, also called *end time*, is simply RE = LE + x.

Compensations StreamInsight allows users to issue compensations (or corrections) for earlier reported events, by the notion of *retractions* [1, 20, 21, 22], which indicates a modification of the lifetime of an earlier event. This is supported by an optional third control parameter RE_{new}, that indicates the new right endpoint of the corresponding event. Event deletion (called a full retraction) is expressed by setting RE_{new}=LE (i.e., zero lifetime).

Canonical History Table (CHT) This is the logical representation of a stream. Each entry in a CHT consists of a lifetime (LE and RE) and the payload. All times are application times, as opposed to system times. Thus, StreamInsight models a data stream as a time-varying relation, motivated by early work on temporal databases by Snodgrass et al. [3]. Table 1 shows an example CHT. This CHT can be derived from the actual physical events (either new inserts or retractions) with control parameter $c = <LE, RE, RE_{new}>$. Table 2 shows one possible physical stream with an associated logical CHT shown in Table 1. Note that a retraction event includes the new right endpoint of the modified event. The CHT (Table 1) is derived by matching each retraction in the physical stream (Table 2) with its corresponding insertion (i.e., matching by event ID) and adjusting the RE point of the event accordingly. StreamInsight operators are well-behaved and have clear semantics in terms of their effect on the CHT. This makes the underlying temporal algebra deterministic, even when data arrives out-of-order.

TABLE I
EXAMPLE OF A CHT

ID	LE	RE	Payload
E0	1	5	P1
E1	4	9	P2

TABLE II
EXAMPLE OF A PHYSICAL STREAM

ID	Type	LE	RE	RE_{new}	Payload
E0	Insertion	1	∞	-	P1
E0	Retraction	1	∞	10	P1
E0	Retraction	1	10	5	P1
E1	Insertion	4	9	-	P2

The *sync time* of a physical event is defined as the earliest time that is modified by the event. For example, the sync time of a physical insert event with lifetime [LE, RE) is LE, while the sync time of a modification event with endpoints LE, RE, RE_{new} is min(RE, RE_{new}).

B. Event Classes

Users can use lifetimes to model different application scenarios. For instantaneous events with no lifetime, RE is set to LE+h where h is the smallest possible time-unit. We refer to such events as *point events*. On the other hand, there may be events that model an underlying continuous signal being sampled at intervals. In this case, each event samples a particular value, and has a lifetime until the beginning of the next event sample. We refer to such events as *edge events*. The most general form of events have arbitrary endpoints depending on when the modelled event came into and went out of existence – these events are referred to as *interval events*. For instance, Figure 2(A) depicts lifetimes of three interval events.

C. Detecting Progress of Time

We need a way to ensure that an event is not arbitrarily out-of-order. The lack of such a facility causes two issues:

- We can never declare any produced output to be "final", i.e., it cannot change due to future events. This declaration of output as final is useful in many cases, e.g., when we need to prevent false-positives in scenarios where correctness is important, such as directing an automatic power plant shutdown based on detected anomalies.
- We cannot clean historic state in the DSMS, since it may be needed forever in order to adjust previous output.

To solve this problem, we need the notion of stream progress, which is realized using time-based punctuations [1, 4, 6]. A time-based punctuation is a special event that is used to indicate time progress. These punctuations are called Current Time Increments (or *CTIs*) in the terminology of StreamInsight. A CTI is associated with a timestamp *t* and indicates that there will be no future event in the stream that modifies any part of the time axis that is earlier than *t*. Note that we could still see retractions for events with LE less than t, as long as both RE and RE_{new} are greater than or equal to *t*.

D. Operator Types

A streaming *continuous query* (*CQ*) consists of a tree of operators, each of which performs some transformation on its input streams and produces an output stream. There are two types of operators: *span-based* operators and *window-based* operators.

1) Span-based Operators

A span-based operator accepts events from an input, performs some computation for each event, and produces output for that event with the same or possibly altered output event lifetime. For example, Filter (see Figure 2 (A)) is a span-based operator that selects events which satisfy certain specified conditions. The lifetime of the output event is equivalent to the entire "span" of the input event's lifetime.

2) Window-based Operators

On the other hand, *aggregation operators* such as Count, Top-K, Sum, etc. work by reporting a result (or set of results) for every unique *window*. The result is computed using all events that belong to that window. Window-based operators in the StreamInsight extensibility framework use a novel notion of windows, which we discuss next.

E. Imposing Windows on Event Streams

The basic idea is that we achieve windowing by simply dividing the underlying time-axis into a set of possibly overlapping intervals, called *windows*. Events are assigned to windows based on a *"belongs-to"* condition. For example, in case of all our window types[1], an event *belongs to* a window if and only if the event's lifetime overlaps with the window's interval (time span).

The desired operation (e.g., sum or count) is applied over every window (i.e., over the set of events belonging to that window) as time moves forward. The output of a window is the computation result of the operator over all events in that window, and has a lifetime that is usually equal to the window duration.

Interestingly, we will see in Section 3 that this core windowing technique can be used to express all common notions of windows, including sliding windows, hopping windows, and count-based windows, by simply varying how the time-axis is divided into intervals.

[1] We will see in Section 3.2 that count-based windows have an added restriction beyond the overlap condition.

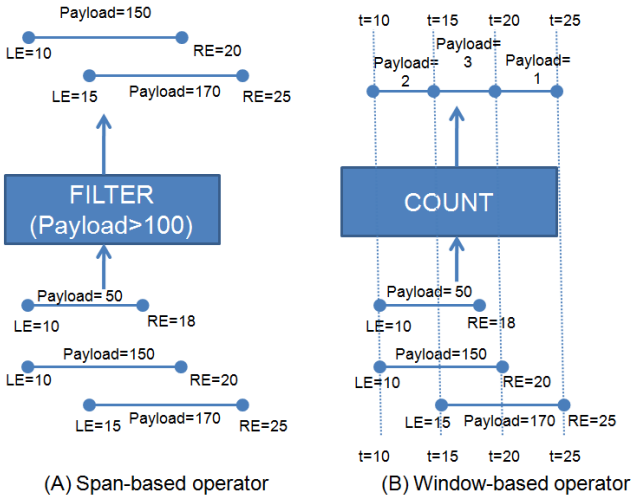

Figure 2. Span-based operators vs. window-based operators.

Figure 2 (B) illustrates the "Count" aggregate over a 5 second *tumbling window*. We see that a tumbling window is "simulated" by simply dividing the time interval into a consecutive sequence of disjoint intervals of equal width (the window length). One output event is produced for every unique window as shown, where the output is computed over all events whose lifetimes overlap with that window. Various types of windows supported by Microsoft StreamInsight are covered in detail in Section 3.

III. The Query Writer's Perspective

As mentioned in Section 1, the central design goals for the query writer are ease-of-use and flexibility. Ease-of-use means that all UDM implementation details are hidden from the query writer. The query writer simply invokes a UDM by its name, possibly passing some initialization parameters if needed. Flexibility means that the query writer can control the UDM behavior via two parameters: (1) the window specification and (2) the input/output timestamping policy. This section covers the language surface area for the query writer to invoke UDMs, with particular focus on the window specification and the input/output timestamping policy parameters.

A. The LINQ Surface Area for Extensibility

On top of the streaming algebra, a suitable query language exposes the operator functionality to the user. A good declarative query language should be a concise, yet intuitive interface to the underlying algebra. In StreamInsight, the Language Integrated Query (LINQ) [5, 18] was chosen as approach to express CEP queries. LINQ is a uniform programming model for any kind of data that introduces queries as first class citizens in the Microsoft .NET framework. StreamInsight supports three fundamental types of user-provided extensions to the system – user-defined functions, user-defined aggregates, and user-defined operators. Based on the type of extension, UDMs surface either as method calls in the case of span-based stream operators, or as extension methods on windowed streams, in the case of window-based stream operators.

1) User-Defined Functions (UDFs): UDFs are method calls with arbitrary parameters and a single return value. By using UDFs, expressions of any complexity are possible. They can be used wherever ordinary expressions (span-based stream operators) occur: filter predicates, projections, join predicates, etc. The method call is evaluated for each event. A user-defined function must be compiled into an assembly that is accessible by the StreamInsight server process at runtime. The example below invokes the *valThreshold* UDF from the MyFunctions library. The UDF is executed for every event *e* in the stream and takes a single value as parameter (i.e., *e.id*).

```
var filtered =
   from e in stream
   where e.value < MyFunctions.valThreshold(e.id)
   select e;
```

2) User-Defined Aggregates (UDAs): A UDA is used on top of a window specification (e.g., hopping, snapshot, or count-based window) to aggregate the events contained in each window. At runtime, a UDA receives a set of events, representing the members of a single window, and produces the aggregation result value that maps to one of the StreamInsight primitive types (e.g., integer, float, string, etc). In the example below, the *median* UDA is invoked for every window *w* to compute the median of the payload field *e.val* over all events with lifetimes that overlap the window *w*.

```
var result = from w in s.HoppingWindow(...)
             select new { f1 = w.Median(e.val) }
```

3) User-Defined Operators (UDOs): Similar to UDAs, UDOs are used on top of a window specification to process the events in each window. However, there are three differences between a UDA and a UDO. First, a UDA returns a value of a primitive type while a UDO returns an entire event payload with one or more field values. Second, a UDA return value that contributes to exactly one output event while a UDO generates zero or more events. Third, the UDO has the option to timestamp its output events. This type of UDO (described in Section 4) is called a *time-sensitive* operator. As an example, a pattern detection UDO may detect zero or more patterns of interest in a single window. The pattern detection UDO generates an output event for every detected pattern. Each output event may consist of one or more payload fields that describe the pattern. Moreover, the UDO decides on how to timestamp each output event. A user defined timestamping of the output becomes crucial in application scenarios where detected patters are not expected to last for the entire window duration. The following example shows how a query writer invokes a UDO.

```
var newstream =
   from w in inputStream.SnapshotWindow(...)
   select w.MyPatternDetectionUDO();
```

B. The Window Specification

The window specification defines the shape of the windows and, consequently, defines the event membership in each window. There are the four different supported window types in StreamInsight: hopping windows, tumbling windows, snapshot windows, and count-based windows.

1) Hopping Windows: Hopping windows divide the timeline into regular intervals, independently of event start or end times. Each hopping window is offset by a hop size w.r.t. the previous window. The window is defined by two time spans: the hop size *H* and the window size *S*. For every *H* time units, a new window of size *S* is created. Figure 3 shows a stream that contains three events: e1, e2, and e3. The vertical bars show a hopping window segmenting the timeline.

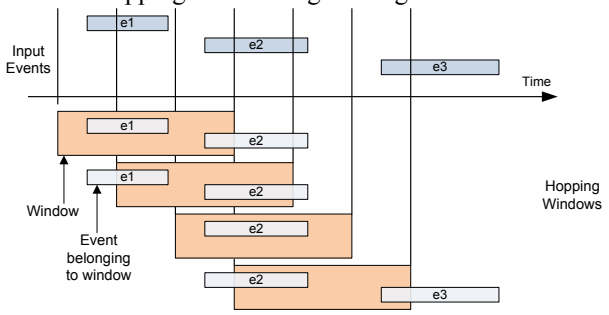

Figure 3: Hopping windows

2) Tumbling Windows: Figure 4 illustrates a special case of the hopping window where the hop size *H* equals the window size *S*. This special case of gapless and non-overlapping hopping window is called the *tumbling window*. Note that for both hopping and tumbling windows, if an event spans a window boundary, the event becomes a member in every window it overlaps. This is the case for events e1 and e2 in Figure 3.

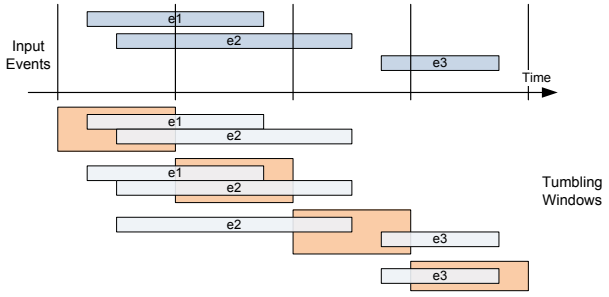

Figure 4. Tumbling windows.

3) Snapshot Windows: Instead of a fixed grid along the timeline, snapshot windows are defined according to the start and end times of the events in the stream. The size and time period of the window is defined only by the events in the stream. A *snapshot* is defined as: the maximal time interval where no change is observed in the input. In other words, it is the maximal time interval that contains no event endpoints (*left endpoint* LE or *right endpoint* RE).

For each pair of consecutive event endpoints (LE or RE), a snapshot window is created. By this definition, all event endpoints fall on window boundaries and never in between. That is, snapshot windows divide the timeline according to all occurring changes in the input that are signaled by event endpoints.

Figure 5 shows a stream with three events: e1, e2, and e3. The vertical bars show the snapshot window boundaries that are defined by these events. Based on the start time and end

time of events, only event e1 is in the first snapshot window. However, both events e1 and e2 are overlapping and, therefore, are included in the second window.

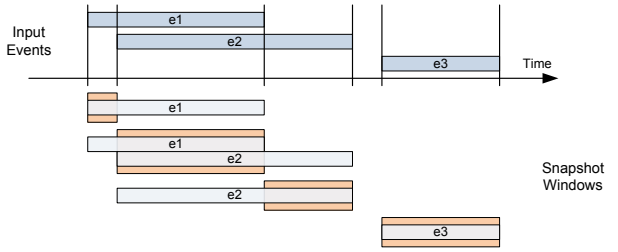

Figure 5. Snapshot windows.

4) Count Windows: A count window with a count of *N* is defined as the timespan that contains *N* consecutive event endpoints. Because StreamInsight supports multiple classes of events (point, interval, or edge as described in Section 2.2), there are two types of count windows supported over interval events. Count by start time windows span *N* event start times (LE). Here, an event *belongs to* a window if its LE is within the window. Similarly, Count by end time windows span *N* event end times (RE). In this case, an event *belongs to* a window if its RE is within the window. We explain the count by start times windows in this section and the generalization count by end time windows is straightforward. Count windows move along the timeline with each distinct event start time. Hence, each new event will cause the creation of a new count window that extends in the future to include *N* event start points in the window, as long as the there are *N* events in the future. If there are less than *N* events, no window is created. If each event on the timeline has a unique timestamp, the number of events in each such window is equal to *N*. In case of multiple events with the same event start time, the number of contained events can be higher than *N*.

Readers may wonder why we choose to count the number of start times instead of the number of events, to define count window. The main reason is that we want our windowing operation to be well-behaved and deterministic. In case we only wanted the most recent *N* events to form part of a window, there can be ambiguity regarding which events form part of the window, in case there are multiple events with the same left endpoint or start time (LE). Our definition of count windows fits well with our overlap-based window semantics, and is also compatible with our customer applications that require count windows.

Figure 5 shows an example count by start time window with *N*=2.

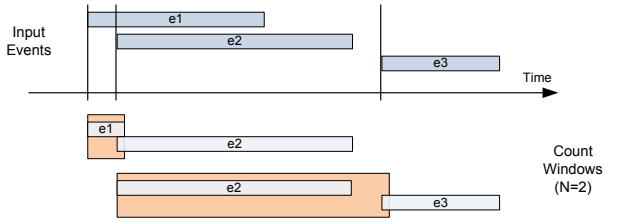

Figure 6. Count window that counts event by start times.

C. The Clipping and Timestamping Policies

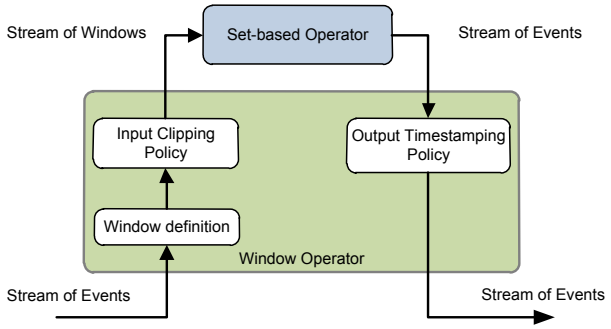

Figure 7. Input clipping and output timestamping policies.

Refer to Figure 7. Apart from the definition of the window (hopping, tumbling, snapshot, or count window), the query author can influence (1) how the windowing operation affects the lifetimes of the input events that are contained in the window before they are passed to the window-based operation and (2) how the lifetimes of the operation's output events are to be adjusted. Both policies are specified by the query writer as part of the window operator to control or override the default timestamps of the input/output events of the window operations. These transformations are called *input clipping policy* and *output timestamping policy*, respectively.

1) Input Clipping Policy

The input clipping policy adjusts the timestamps of the incoming events with respect to the window boundary. There are four clipping policies for input events:

- ***Left Clipping***: Clips the event's left endpoint e_{LE} to the window left boundary w_{LE} if the event *e* starts before the window start time (i.e., if $e_{LE} < w_{LE}$, set $e_{LE} = w_{LE}$).
- ***Right Clipping***: Clips the event's right endpoint e_{RE} to the window right boundary w_{RE} if the event *e* ends after the window end time (i.e., if $e_{RE} < w_{RE}$, set $e_{RE} = w_{RE}$).
- ***Full Clipping***: Clips event from both sides by applying both left and right clipping policies.
- ***No Clipping***: Events are sent to the UDM without being clipped.

As a consequence of such adjustments, any time-sensitive window operation [2] that takes into consideration the event timestamps while computing the output value is affected by the specified clipping policies. Figure 8 shows how events in a tumbling window are fully clipped to the windows.

The decision to clip an event with respect to the window boundary has to be based on the operator semantics. For example, a pattern operator that detects the pattern "A followed by B" requires the original event start times to reason about the chronological order of events, and hence cannot work with left clipping if it needs to be able to

[2] An example of time-sensitive user defined modules is given in Section 4.

incorporate the effect of overlapping events that start earlier that the left endpoint of the window.

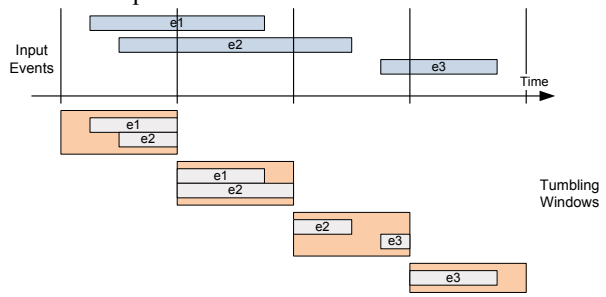

Figure 8: Tumbling windows with fully clipped events.

Moreover, the right clipping policy has a crucial impact on the progress of output time and on the system resources. Hence, the clipping decision needs to be taken carefully by the query writer. To explain the effect of right clipping on the progress of output time, assume an event e that extends t time units after the window right endpoints (i.e., $e_{.\mathrm{RE}} = w_{.\mathrm{RE}}+t$). Because the output of the window w depends on the $e_{.\mathrm{RE}}$, which extends t time units in the future, and because retractions are expected to change the event's $e_{.\mathrm{RE}}$, the system will not be able to finalize the window output until a *CTI* (Current Time Increment as explained in Section 2) is signaled t time units beyond the window $w_{.\mathrm{RE}}$. Although the system generates speculative output from window w as soon as an event that overlaps the window w is received, the system delays the propagation of the *CTI* signal to the output by t time units. Also, the memory resources taken by the window are not reclaimed till the *CTI* passes $w_{.\mathrm{RE}}$ by t time units. Therefore, for workloads with long living events, right clipping is highly recommended for the liveliness and the memory demands of the system. We discuss liveliness and cleanup in detail in Section 5.

2) Output Timestamping Policy

On the output side, the output timestamping policy decides on the default management of the event's timestamps and their lifetimes. The output of a window based operation is fed to the next operator in the query plan or directed to the consumer of the query output (if it is the last operator in the query plan). There are several possible output timestamping policies:

- Align events to the window boundaries. This policy is the only option for *time-insensitive* operations that do not timestamp their output events (this will be covered in Section 4). Also, it gives the query writer the ability to override the UDM timestamping policy and revert to a default timestamping policy.
- Keep the timestamps and lifetimes of output events unchanged. This policy applies only to time-sensitive UDMs that timestamp the output events. The only restriction on that policy is that a UDM is not allowed to generate an output event in the past ($e_{.\mathrm{LE}} < w_{.\mathrm{LE}}$). Past output is vulnerable to cause *CTI* violation, that is, generate output after time progression guarantees has been established on the output.
- Clip the events to the window boundaries. This policy keeps the timestamps and lifetimes of output events as indicated by the UDM. However, if the event stretches beyond the window boundary, it gets clipped to the window boundary.

We discuss the effect of these timestamping policies on liveliness and state cleanup in detail in Section 5, and present a new policy that can achieve maximal liveliness by imposing a new restriction the lifetime of events produced by the UDM.

IV. The UDM Writer's Perspective

The extensibility framework has been designed with attention given to two classes of UDM writers. The first class represents a wide range of software vendors who are *not* trained to think under the data streaming model with its temporal dimension. Moreover, this class of UDM writers has developed libraries of UDMs over years of experience in their domain. The second class of UDM writers targets stream-oriented applications where temporal attributes are first class citizens in their business logic.

The first class of users, call them *traditional users*, are interested in porting their logic from traditional databases to the streaming world with minimal efforts. According to the "portability and compatibility design principle" (refer to Section 1), StreamInsight conveniently accommodates that class of UDM writers through:

- Preserving a relational view of the data.
- Managing the temporal dimension on behalf of the UDM writer.
- Handling imperfect event delivery represented by late event arrivals on behalf of the UDM writer

The second class of users, call them *power users*, seeks full control over the temporal attributes of events as well as maximum achievable performance. According to the "powerful time management design principle" (as described in Section 1), StreamInsight conveniently accommodates that class of UDM writers through:

- Authorizing UDMs to read the temporal dimension of input events (LE and RE)
- Giving UDMs the ability to timestamp their output events.
- Supporting the incremental evaluation of output results.

From all three extensibility interfaces (UDFs, UDOs and UDAs), UDFs are the most straightforward to implement. A UDF is defined as a .NET method call with an arbitrary number of parameters. The UDF is a span-based operation that is evaluated for each event. For brevity, we focus our discussion in this section on UDOs and UDAs that are executed over a window-based model. As explained in Section 3, a UDO or UDA is called for each window at runtime.

To cover the two classes of users, StreamInsight requires the UDM writer to take two decisions in advance:

The first decision lets the UDM writer declares his model of thinking. StreamInsight supports two models of thinking: incremental and non-incremental models. Non-incremental

model provides a relational view of the world to UDM writers while the incremental model provides the deltas or the changes in the input to the UDM since the UDM's last invocation. The second decision indicates whether the UDM is time sensitive or time insensitive. Time insensitive UDMs deal with *payloads*. However, time-sensitive UDMs handle *events* (i.e., payloads plus temporal attributes). Time sensitive UDMs read the temporal attributes of input events, reason about time, generate and timestamp output events.

In the following subsections, we describe non-incremental versus incremental UDMs as well as time sensitivity in UDMs.

A. Non-incremental vs. Incremental UDMs

1) Non-Incremental UDMs

A UDA/UDO can be implemented as a non-incremental operation. The engine passes a list (or *IEnumerable* according to the .NET framework terminology, or a table in the relational database terminology) of all events that overlap the window to the UDM. Thus, according to the portability and compatibility design principle, the UDM writer adopts a relational viewpoint towards the set of input events and defines the operation on the set in a declarative way. Figure 9 illustrates that a UDM writer is expected to implement a single method, called *ComputeResult*. The *ComputeResult* method accepts an IEnumerable of payloads (in case of time-insensitive UDM) or an IEnumerable of events (in case of time-sensitive UDM). The *ComputeResult* method performs the computation on the given input and generates a scalar field value (in case of a UDA), an IEnumerable of payloads (in case of time-insensitive UDO), or an IEnumerable of events (in case of time-sensitive UDO). Note that the UDM writer deals with payloads or events (payloads plus temporal attributes). The UDM writer does not have to worry about imperfections in event delivery. Section 5 (Systems Internals) provides the details on how the underlying framework invokes the UDM as many time as the number of windows and as many time as needed to process insertions and retractions.

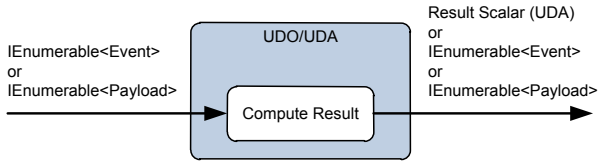

Figure 9. Non-incremental UDMs.

2) Incremental UDMs

In order to tap the full benefit of a streaming system, we need a mechanism to allow advanced UDM writers to express *incremental* computations, such that the framework maintains a state per window that is updated incrementally with the arrival of every insertion or deletion event. Figure 10 shows that the UDM writer is expected to implement three methods. The *AddEventToState* takes as input the state of a window and the set of *delta* events that overlap that window. *Delta* events are the events that have arrived to the system and overlap window *w* since the last invocation of the UDM over window *w*. The *AddEventToState* incrementally updates the state of the window to reflect the effect of adding the delta events to the state. Similarly, the *RemoveEventToState* incrementally updates the state of the window to reflect the effect of removing the delta events on the state. The UDM writer also implements the *ComputeResult* method that computes the output given the window state as input. Based on the incoming insertion, retractions and *CTIs*, the underlying extensibility framework (as explained in Section 5) invokes the UDM methods to incrementally maintain a per-window state and to generate the proper insertions and retraction to the output.

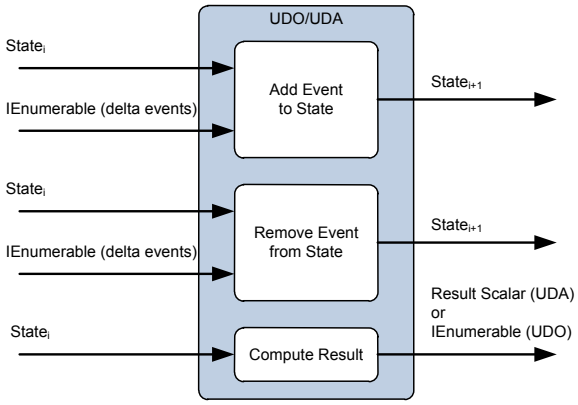

Figure 10. Incremental UDMs.

B. Time Sensitivity in UDMs

1) Time-Insensitive UDMs

Time insensitive UDMs do not consider the temporal dimension while computing the result. Many time-insensitive UDMs have been written over the years for traditional databases. These UDMs are still of value and are expected to be executed on sets of event payloads in the streaming domain.

2) Time-Sensitive UDMs

Time sensitive UDMs are classified into two groups: the first group of time-sensitive UDMs is concerned about the input's temporal attributes. The input's temporal attributes are (1) lifetime of input events and (2) the duration of the window that contains the set of input events, i.e., the window descriptor. The second group of time-sensitive UDMs is concerned with the output's temporal attributes, that is, the timestamps of the output events. The two groups are not mutually exclusive. Some UDMs read the input temporal attributes and generate the timestamps of the output events according to the UDM logic.

C. Example End-to-End UDM Development

In this subsection, we provide an example of a simple time-insensitive user defined aggregate (*MyAverage*). Then, we leverage the aggregate with reasoning about the temporal dimensions to provide a time-weighted average version of the aggregate (*MyTimeWeightAverage*). *MyTimeWeightAverage* adjusts the contribution of each element to the computed

overage by the event's lifetime compared to the entire window duration.

Example. The following code snippet shows how a simple time-insensitive aggregate (*MyAverage*) that computes the average over a given payload field. *MyAverage* does not reason about temporal properties of incoming events. *MyAverage* derives from the *CepAggregate* base class, which is a system-provided base class, and declares the input and output types to be of double data type. *MyAverage*.*GenerateOutput* method accepts an IEnumerable of payloads that are of type double and returns the average as single value of type double that represents the average over the given IEnumerable of payloads.

```
public class MyAverage :
             CepAggregate<double, double>
{
    public override double
    ComputeResult(IEnumerable<double> payloads)
    {
        return events.Sum() / events.Count();
    }
}
```

In order to integrate the user-defined aggregate into the LINQ development experience for the query writer, an extension method needs to be defined:

```
static public class UDAExtensionMethods
{
    [CepUserDefinedAggregate(typeof(MyAverage))]
    public static double
    MyAverage(this CepWindow<T> window,
            Expression<Func<T, double>> map)
    {
        throw CepUtility.DoNotCall();
    }
}
```

As shown in this example, the extension method serves the purpose of associating the aggregate definition with a LINQ extension method signature. The extension method is never actually executed but is used by the LINQ provider to insert the proper method call into the runtime query plan. The signature of the extension method includes an expression that is used by the query writer to map the stream's input event type to the UDM expected input data type. Note that UDMs are pre-packed modules that operate on payload of type *T*. The mapping expression bridges the gap between the incoming events' schema and the UDM expected payload type *T*. In the above example, the mapping expression is expected to pick a payload field of type double from the input event schema.

In contrast to time insensitive UDMs, a time-weighted average is a time-sensitive aggregate that reasons about the event lifetimes w.r.t. the window time span:

```
public class MyTimeWeightedAverage :
     CepTimeSensitiveAggregate<double, double>
{
    public override double
    ComputeResult(
       IEnumerable<IntervalEvent<double>> events,
       WindowDescriptor windowDescriptor)
    {
        double avg = 0;
        foreach (IntervalEvent<double>
                 intervalEvent in events)
        {
            avg += intervalEvent.Payload *
                 (intervalEvent.EndTime-
                 intervalEvent.StartTime).Ticks;
        }
        return avg / (windowDescriptor.EndTime -
                windowDescriptor.StartTime).Ticks;
    }
}
```

MyTimeWeightAverage class derives from the CepTimeSensitiveAggregate base class that is provided by the system to declare the time sensitivity of the UDM. As shown by the above code snippet, *MyTimeWeightAverage* reads the window's temporal properties (*windowDescriptor*.*StartTime* and *windowDescriptor*.*EndTime*) as well as the input event life times (*intervalEvent*.*StartTime* and *intervalEvent*.*EndTime*). Note that time sensitive UDMs have the option to timestamp their output event. If the UDM does not timestamp the output, the output events are by default timestamped with the entire window duration timestamps ($e_{.LE}$ = *windowDescriptor*.*StartTime* and $e_{.RE}$ = windowDescriptor.*EndTime*).

V. System Internals

We now discuss the system internals for UDMs, i.e., we discuss how StreamInsight efficiently handles incoming events (insertions and retractions) and windows internally, in order to invoke the appropriate API calls for the UDMs and produce output events. UDFs are easy to handle; for each incoming event, the system first evaluates each UDF input parameter from the event content, and then invokes the user-defined function. The result of the user-defined function is returned back to the system to continue normal processing of the remainder of the plan. We focus on UDOs/UDAs in the rest of this section.

A. Invoking User APIs

The system accumulates a set of events for a window and invokes the appropriate user API call, optionally modifying the lifetimes of the event(s) sent to the UDO based on a user-specified *input clipping policy* (see Section 3.3.1). The UDO produces a set of results. There are two possible modes:

If the user is using the time-insensitive API, they return a set of payloads, and the UDO adds timestamps to the rows in order to convert them into events. Timestamp addition is based on the *output timestamping policy* (see Section 3.3.2), which may be specified by the user. The only option for time-insensitive UDOs is to set the output lifetime equal to the window lifetime.

If the user is using a time-sensitive API, they return events with timestamps directly to the system. The UDO may specify an *output timestamping policy* that adjusts or restricts the possible lifetimes that the UDO can assign to its output events.

B. Definitions

Recall that each unique window W corresponds to a time interval, with left and right endpoints (W.LE and W.RE). The window is associated with all events whose lifetimes overlap the interval [W.LE, W.RE). We define the current *watermark*

m as the maximum of (1) the latest received CTI and (2) the maximum LE across all received events.

We say that a window is *closed* if no future event can affect (overlap) that window. This is determined based on received CTIs which prevent events from occurring arbitrarily in the past. Otherwise, the window is said to be *active*.

C. Data Structures

Our algorithms maintain the invariant that output is produced for all non-empty windows that do not overlap the interval $[m, \infty)$. CTI handling is discussed in Section 5.5.

Refer to Figure 11. We maintain two data structures in the system:

WindowIndex: This data structure tracks all active windows in the system. It is organized as a red-black tree, with one entry for each unique window (the window could be of any type as described in Section 3). Each entry for window W is indexed W.LE. Each window entry contains (1) *W.#endpts*, the number of event endpoints within the window and (2) *W.#events*, the number of events that overlap the window.

EventIndex: This data structure tracks all active events (i.e., events that have not been cleanup up by CTIs). It is organized as a two-layer red-black tree, where the first layer indexes events by RE and the second layer indexes events by LE. Note that we could also use an *interval tree* to replace this data structure.

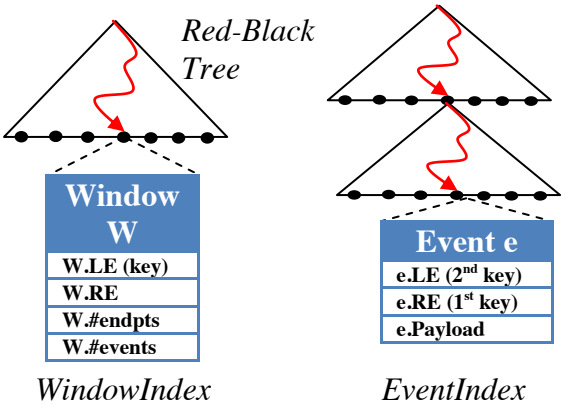

Figure 11: WindowIndex and EventIndex data structures.

D. Non-incremental UDOs

At an abstract level, the algorithms for both incremental and non-incremental UDOs have four phases:

Determine which existing windows are affected as a result of the incoming event.

Issue retractions for affected windows to delete the older output produced for those windows

Update data structures, to take the incoming event into account. This can cause new windows to be created or existing windows to be deleted or merged.

Issue new events as output, for all affected windows (after the data structure is updated).

Determine Affected Windows When a new event e arrives at the operator, we first determine which windows are *affected* by the insertion, i.e., their event set membership would change as a result of the insertion. In case of an insertion, this is exactly all windows whose lifetimes overlap [e.LE, e.RE). In case the event is a lifetime modification, the set of affected windows is determined only based on the windows that overlap the changed part of the event lifetime which is [min(RE, RE_{new}), max(RE, RE_{new})). Note that in case of count windows, we perform post-filtering to ensure that our modified belongs-to relation is satisfied.

Issue Full Retractions For each affected window W, we use EventIndex to retrieve all (old) events that overlap [W.LE, W.RE). Finally, the user API is invoked for the window (with the old event set), to produce a set of events. The system issues *full retractions* for all these events.

Note that the interface between the system and the UDO is stateless, hence we needed to invoke the UDO again to determine what events it produced earlier, so that those events can be retracted appropriately. Thus, a UDO needs to ensure that it is *deterministic*, i.e., two invocations of the API with the same input will cause the same output to be produced.

Update Data Structures Next, we update WindowIndex to take the newly inserted event into account. This may cause a new window to be created or existing windows to be split. Further, the counters W.#endpts and W.#events are updated. In case of a lifetime modification event, W.#endpts may become 0 for any window W; as a result, the window W is deleted. An event lifetime modification can cause existing windows to be merged or deleted. We finally insert event e into EventIndex to complete updating data structures.

Produce Output Events Finally, for every new window affected by the event, we again invoke the user API to produce a set of events. These are output directly to the end-user as insert events. We follow *empty preserving semantics*, where a window that contains no overlapping events (i.e., W.#events=0) does not produce any ouput by default.

E. Incremental UDOs

An incremental UDO exposes an API that allows incremental operations over a pre-defined *operator state*. The system maintains the state for each window (as an opaque object) on behalf of the UDO. Thus, each entry in WindowIndex maintains window state as an additional piece of information.

The algorithms are similar to the non-incremental case, with some modifications as follows. When an event arrives at the operator, for each affected window, we invoke the UDO with the old state (instead of the entire set of old events) to produce the set of events to be fully retracted. After updating the data structures as before, we invoke the UDO for each affected window, by passing it the old state and the new event – the UDO processes this input incrementally and returns the new state and output events/payloads. WindowIndex is updated with the new state, and the events/payloads are processed as before.

F. CTI Handling

CTIs (punctuations) are used by the system to produce output (if the CTI increases the current watermark) as well as to clean up operator state in the data structures.

1) UDO Liveliness with CTIs

In the most general form time-sensitive UDOs with no restrictions on the lifetime of events produced by the UDO, we can *never* issue CTIs as output because any window could potentially produce an output event with LE=-∞. Since this is unacceptable, we now propose specific restrictions on UDOs that can improve liveliness.

As a first step, assume that we enforce the following restriction on output intervals: a time-sensitive UDO invoked for window W may produce an event e only if e.LE ≥ W.LE. In other words, the output timestamp must be within or after the current window. We refer to this particular *output timestamping property* as *WindowBasedOutputInterval*. Recall from Section 3.3.2 that we can enforce this policy by clipping output event lifetime to the window boundaries. This restriction gives us a limited amount of liveliness. Specifically, let c be the latest CTI received on the operator's input. Let W be the earliest window associated with an event e_1 such that e_1.RE>c. With only the above restriction, when we use a time-sensitive UDO, we will never be able to issue an output CTI with timestamp > W.LE. To see why, note that since we have a CTI only until timestamp c, e_1 can still change in the future, i.e., its RE can increase to a larger value. This change to e_1 results in a change to the set of events in W, and causes W to be recomputed, potentially producing an output event with timestamp W.LE.

In the special case of a window W having an event with infinite lifetime, we can never issue a CTI beyond W.LE. To remedy this, a further improvement to liveliness is possible by setting the *input clipping policy* (see Section 3.3.1) such that the RE of events belonging to a window W are clipped to W.RE before being sent to the UDO. For many UDOs such as time-weighted average, this is an acceptable restriction because they do not care about the actual RE of the event if the event RE is beyond W.RE. Under this clipping policy, we can propagate a CTI until W.RE, where W is the latest window such that c ≥ W.RE. To see why this is the case, we note that the clipped version of events belonging to window W (or earlier) can never change because the CTI has moved beyond W.RE.

Improving Liveliness Further Even with the above improvement, there is no guarantee that when new a CTI is received, a new output CTI with the increased timestamp will be output in response. We fix this problem by defining a new *output timestamping policy* called *TimeBoundOutputInterval*. The basic idea is that given a window W into which a physical event e (insertion or retraction) is being incorporated, the output event(s) produced by the UDO in response to that event are constrained to have LE ≥ sync time of e (see Section 2.1 for the definition of sync time). Most common UDOs including time-weighted average, pattern matching, traditional aggregates, and top-k are *time-bound*, i.e., they adhere to the *TimeBoundOutputInterval* restriction. It is easy to see that with this restriction, we can propagate CTIs with maximal liveliness, i.e., whenever there is an incoming CTI with timestamp c, we can produce an output CTI with timestamp c.

2) Internal State Cleanup using CTIs

Beyond ensuring liveliness, an important use of CTIs is state cleanup. We need to get rid of old entries from our data structures as soon as they are not needed, so that memory is freed up for new events and other operators in the system.

When we receive a CTI with timestamp c, we prune WindowIndex to get rid of *closed windows*, i.e., windows that will never be required by the system in the future. The process is identical for incremental and non-incremental UDOs. There are three cases:

If the UDO is time-insensitive, we can delete a window W as soon as W.RE ≤ c. This is because future lifetime modifications of events in this window are guaranteed not to affect this window.

If the UDO is time-sensitive with no input event clipping, we can prune WindowIndex by deleting all windows W that are closed, i.e., every event e belonging to W has e.RE ≤ c. This is a necessary condition because any future event lifetime modification could change the window definition, causing this window to be re-computed. This requirement can cause windows to be alive for a long time if events have long lifetimes. We can alleviate this problem by specifying the input event clipping policy, as described next.

If the UDO is time-sensitive with input event clipping, i.e., we clip the RE of input events to the window boundary, we can delete a window as soon as W.RE ≤ c. This is possible because in this case, even if an event lifetime changes, its clipped version that lies within W will not change, and hence W will not require re-computation.

We also have to delete events (and corresponding entries) from the EventIndex that are no longer required. Deletion of events is simple: we delete all events that belong only to closed windows (see above for what constitutes a closed window under various circumstances).

VI. Experiences and Lessons Learned

We have implemented and deployed the extensibility framework in Microsoft StreamInsight, and have received valuable feedback from our customers. In this section, we summarize our experiences and user-feedback received during this process of extending a complex event processing system with powerful yet easy-to-use user-defined capabilities.

A. Temporal Algebra

StreamInsight is based on a clean well-defined and deterministic temporal algebra and operator semantics. Our past experience has shown that clean semantics, which are a cornerstone of traditional databases and relational algebra, are necessary for meaningful operator composability, repeatable behavior, query debuggability, and cost-based query optimization. This foundation helped in building a clean extensibility solution without making operational decisions. Our algebra helped clearly understand and control the implications of physical effects such as CTIs, disorder, retractions, etc. and tackle the core underlying issues behind

subtle aspects such as liveliness and state cleanup. Our novel semantics for count-based windows were also driven by the need for determinism.

B. Liveliness

As mentioned above, liveliness turned out to be more subtle than expected. We initially expected that this would not causes issues beyond those seen with internal operators, but finally it turned out that we had to spend significant time working out a practically usable and efficient solution.

C. Window Definitions

We had an initial solution for windowing that simply modified event lifetimes in order to simulate windowing, but quickly found that this was inadequate for most customer applications, since it destroyed the user's notion of lifetime just for the purpose of introducing windowing.

D. API Interfaces

We found that there is a wide range of potential UDM writers, depending upon the age of the application and the specific requirements of the business domain. Thus, simply providing one powerful API to all UDM writers was not acceptable, due to the broad range of development expertise and requirements in the community.

E. Role Separation

The separation of roles of query writer from the UDO writer was found to be crucial in a practical sense, in order to make the solution marketable in business environments, where possibly different departments and users with widely different backgrounds and expertise assume these roles.

F. Programming Interface

LINQ as an end-user programming surface has been generally well-received, as query writers do not have to worry about "hiking" the query over the client/server wall. On the other hand, some query writers prefer a SQL-style interface, for which we are looking at alternatives such as extending languages such as StreamSQL [23]. One inconvenience with our current extensibility framework is that the UDM module itself needs to be hiked over the wall to be made accessible to the StreamInsight server – we are looking at alternatives to handle this in a cleaner way, such as UDM serialization.

VII. CONCLUSIONS

Leveraging a data streaming system with the ability to host and execute user-defined modules (UDMs) enables domain experts to extend and deploy the system in multiple environments. This paper presented the extensibility framework in Microsoft StreamInsight from three perspectives.

The first perspective is a query writer's perspective. Because a single UDM can be invoked by hundreds of queries, the role of the query writer is designed to be as simple as invoking a method call and as flexible as executing the same UDM under different window specifications and different input/output policies.

The second perspective is the user defined module (UDM) writer's perspective. Simple users get the ability to write powerful UDMs without the need to worry about steam-specific event types (i.e., insertion, retractions) or temporal attributes of events (event timestamp, and duration). More interestingly, UDMs writers get the ability to port libraries they have created for traditional database systems with minimal amount of effort. Meanwhile, advanced UDM writers get the flexibility to manage the temporal aspect of input and output events within their code.

Finally, the paper presented the extensibility framework from a system internals perspective. This perspective covers how the system hosts a UDM, how the system relieves the UDM writer from stream-specific operations, and how it carries over time management and event imperfection handling on behalf of the UDM writer. This perspective also addresses how the system shoots for several optimization opportunities while executing users' code.

REFERENCES

[1] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. *Consistent Streaming Through Time: A Vision for Event Stream Processing*. In Proceedings of CIDR, 412-422, 2007.

[2] Jonathan Goldstein, Mingsheng Hong, Mohamed Ali, and Roger Barga. *Consistency Sensitive Streaming Operators in CEDR*. Technical Report, MSR-TR-2007-158, Microsoft Research, Dec 2007.

[3] C. Jensen and R. Snodgrass. *Temporal Specialization*. In proceedings of ICDE, 594-603, 1992.

[4] Utkarsh Srivastava, Jennifer Widom. *Flexible Time Management in Data Stream Systems*. In PODS, 263-274, 2004.

[5] Paolo Pialorsi, Marco Russo. *Programming Microsoft LINQ*, Microsoft Press, May 2008.

[6] Peter Tucker, David Maier, Tim Sheard, Leonidas Fegaras: *Exploiting Punctuation Semantics in Continuous Data Streams*. IEEE TKDE 15(3): 555-568 (2003).

[7] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter Tucker: *Semantics and Evaluation Techniques for Window Aggregates in Data Streams*. SIGMOD 2005: 311-322.

[8] Arvind Arasu, Shivnath Babu, Jennifer Widom: *CQL: A Language for Continuous Queries over Streams and Relations*. DBPL 2003: 1-19.

[9] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, Oliver Spatscheck: A Heartbeat Mechanism and Its Application in Gigascope. VLDB 2005: 1079-1088.

[10] Jin Li et al.: *Out-of-order Processing: A New Architecture for High-Performance Stream Systems*. PVLDB 1(1):274-288 (2008).

[11] M. Ali et al.: *Microsoft CEP Server and Online Behavioral Targeting*. VLDB 2009 (demonstration).

[12] Sankar Subramanian et al.: *Continuous Queries in Oracle*. VLDB 2007: 1173-1184.

[13] D.Abadi et al. The design of the Borealis stream processing engine. In CIDR, 2005.

[14] Oracle Inc. http://www.oracle.com/.

[15] StreamBase Inc. http://www.streambase.com/.

[16] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly Progress Detection in Iterative Stream Queries. In VLDB, 2009.

[17] Moustafa A. Hammad et al.: Nile: A Query Processing Engine for Data Streams. ICDE 2004: 851.

[18] Microsoft LINQ. http://tinyurl.com/42egdn.

[19] Chart Patterns. http://tinyurl.com/6zvzk5.

[20] M. Liu et al. Sequence pattern query processing over out-of-order event streams. In ICDE, 2009.

[21] R. Motwani et al. Query processing, approximation, and resource management in a DSMS. In CIDR, 2003.

[22] E. Ryvkina et al. Revision processing in a stream processing engine: a high-level design. In ICDE, 2006.

[23] N. Jain et al. Towards a Streaming SQL Standard. In VLDB, 2008.