SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2194

Generic windowing support for extensible stream processing systems

Buğra Gedik*,†

Computer Engineering Department, Bilkent University, Ankara 06800, Turkey

SUMMARY

Stream processing applications process high volume, continuous feeds from live data sources, employ data-in-motion analytics to analyze these feeds, and produce near real-time insights with low latency. One of the fundamental characteristics of such applications is the on-the-fly nature of the computation, which does not require access to disk resident data. Stream processing applications store the most recent history of streams in memory and use it to perform the necessary modeling and analysis tasks. This recent history is often managed using *windows*. All data stream management systems provide some form of windowing functionality. Windowing makes it possible to implement streaming versions of the traditionally blocking relational operators, such as streaming aggregations, joins, and sorts, as well as any other analytic operator that requires keeping the most recent tuples as state, such as time series analysis operators and signal processing operators.

In this paper, we provide a categorization of different window types and policies employed in stream processing applications and give detailed operational semantics for various window configurations. We describe an extensibility mechanism that makes it possible to integrate windowing support into user-defined operators, enabling consistent syntax and semantics across system-provided and third-party toolkits of streaming operators. We describe the design and implementation of a runtime windowing library that significantly simplifies the construction of window-based operators by decoupling the handling of window policies and operator logic from each other. We present our experience using the windowing library to implement a relational operators toolkit and compare the efficacy of the solution to an earlier implementation that did not employ a common windowing library. Copyright © 2013 John Wiley & Sons, Ltd.

Received 5 November 2012; Revised 14 February 2013; Accepted 1 March 2013

KEY WORDS: data stream processing; windowing semantics; windowing library

1. INTRODUCTION

Stream processing applications process high volume, continuous feeds from live data sources, employ data-in-motion analytics to analyze these feeds, and produce near real-time insights with low latency. With the explosion in the amount of data available as live feeds, *stream computing* has found wide applications in areas ranging from telecommunications to healthcare to cyber security. The high *volume* of the data to be processed, the *velocity* at which the results need to be produced, and the *variety* of the data sources involved make stream processing applications unique and challenging. To address these challenges, many stream computing platforms have been developed over the last decade, providing languages and runtime systems for developing and deploying stream processing applications effectively. Examples are many both in academia [1–3], as well as in industry [4–6].

One of the fundamental characteristics of stream processing applications is the on-the-fly nature of the computation, which does not require access to disk resident data. This distinguishes stream

*Correspondence to: Buğra Gedik, Computer Engineering Department, Bilkent University, Ankara 06800, Turkey.
†E-mail: bgedik@cs.bilkent.edu.tr

processing systems from traditional databases that adopt the store-and-then-process model of computation. Stream processing applications store the most recent history of streams in memory and use it to perform the necessary modeling and analysis tasks in an online manner. This recent history is often managed using *windows*. All data stream management systems provide some form of windowing functionality. Windowing makes it possible to implement streaming versions of the traditionally blocking relational operators, such as streaming aggregations [7], joins [8], and sorts.

One of the goals of stream computing platforms is to be general enough so that multi-modal applications that process data sources of different nature such as text, video, and structured data can be supported [9]. Furthermore, specific application areas require domain-specific analytic operators that are specialized for the needs of the particular domain at hand [10]. As a result, some of the stream processing languages and frameworks allow user-defined operators, such as SPL [11] in System S [3] and the Microsoft.NET framework LINQ in StreamInsight [12]. For instance, the SPL language, which we use as a case study in this work, makes no differentiation between the relational operators provided by the streaming platform versus the operators that can be developed by third parties as *toolkits*. As such, it provides a set of generic capabilities to user-defined operators, including the ability work with any type (i.e., type genericity), support variable number of ports, define *windows*, take expressions as parameters and accept output assignments, and provide custom aggregation functions to name a few (see [5] for details).

In the presence of user-defined operators as first class citizens, an important issue is the support for windowing, which is the focus of this paper. In particular, there are three fundamental challenges. First, syntactic uniformity should be achieved across different operators while supporting operators with differing levels of windowing support in terms of the window types, policies, and configurations they allow. Second, semantic uniformity should be achieved so that the same syntax has a consistent meaning across different operator invocations. Last, windowing support in stream processing systems become very complex because of a wide variety of window configurations used to support a large number of use cases. It is a requirement for an extensible stream processing system to provide runtime implementation support for windowing that is exposed to the operator developers. However, user-defined operator developers (akin to library developers in general purpose programming languages) should be shielded from the details of windowing policy implementations and instead focus on analytic logic implemented by their operators.

In this paper, we provide a categorization of different window types and policies employed in stream processing applications and give detailed operational semantics for various window configurations. This addresses the semantic uniformity challenge. We describe the design of a windowing library that encompasses a compile-time component used to provide flexible windowing syntax support for user-defined operators, as well as a runtime component used to provide comprehensive implementation support for all window configurations. The compile-time component addresses the syntactic uniformity challenge while allowing operators to define the level of windowing support they aim to provide. The windowing library provides runtime support that decouples plumbing related to window handling from the details of the operator logic. The former is completely handled by the windowing runtime, whereas the latter is implemented by the operator developer by reacting to events from the windowing runtime. In summary, the windowing library runtime adopts a *policy-based design* used to configure windows and an *event-based interface* used to decouple windowing logic from the operator logic. This addresses the final challenge of saving the operator developers from having to deal with the details of window handling.

As part of our categorization of windows, we discuss *tumbling*, *sliding*, and *partitioned* windows. Partitioned windows play a key role in performing *multiplexed processing* [13] in stream processing applications, which in turn forms the cornerstone of stateful parallelism. We introduce policies that define the *eviction* and *trigger* behavior of windows, including *count*, *time*, *attribute-delta*, and *punctuation*-based policies. Punctuations [14] are out-of-band signals carried within streams. We introduce *window punctuations* to communicate window boundaries across operators in a flow graph. We describe punctuation modes for input and output ports of operators and punctuation propagation rules to maintain punctuation semantics in the presence of composition. Finally, we look at *partition eviction* policies used to manage unbounded size growth issues in partitioned windows.

DOI: 10.1002/spe

To the best of our knowledge, the window variations presented in this work cover all variations presented in the literature as part of stream processing systems. We also cover variations that are not discussed elsewhere. A detailed comparison is given in Section 7.

We describe the interface and implementation of our runtime library, which is designed as a container that is configurable with windowing policies and exposes an event-based interface to execute user-defined actions to facilitate the implementation of operator logic. As part of our discussion, we cover topics such as threading in the presence of time-based policies, user-defined policies for selecting partitions to evict, and efficient implementation of incremental computations through *window summarizers*. We also describe our compile-time library and how it provides flexible windowing syntax while separating the implementation decisions from the syntax using compile-time introspection and code generation techniques.

We present our experience using the windowing library to implement a relational operators toolkit in System S [3] and compare the efficacy of the solution to an earlier implementation that did not employ a common windowing library.

In summary, this paper makes the following contributions:

- We provide a categorization of different window types and policies and give detailed operational semantics for various window configurations.
- We describe an extensibility mechanism that makes it possible to integrate windowing support into user-defined operators, enabling consistent syntax and semantics across different operators.
- We describe the design and implementation of a windowing library that significantly simplifies the construction of window-based operators by decoupling windowing and operator logic from each other.
- We present our experience using the windowing library in an industrial strength stream processing middleware.

The rest of the paper is organized as follows. We provide relevant background on SPL and System S in Section 2. We formalize windowing concepts in Section 3. In Section 4, we discuss the runtime design and implementation of our windowing library. We describe our extensibility mechanism in Section 5. Our experience using the library and its evaluation is presented in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2. BACKGROUND

In this section, we provide a brief overview of the basic concepts associated with stream processing applications, using SPL [11] as the language of choice. We also briefly describe the runtime execution of SPL applications within System S.

2.1. Stream processing applications in SPL

Stream processing applications in SPL are expressed as data flow graphs. An SPL application is composed of *operator instances* connected to each other via *stream connections*. An operator instance is a vertex in the application's data flow graph, and a stream connection is an edge. An operator instance is always associated with an *operator*. An operator is a reusable stream analytic. As an example, a `Join` operator may represent a streaming relational join. Different instances of a `Join` operator can process different types of streams, have different kinds of match conditions, or even have different number of outputs (e.g., an outer join may have extra output ports for non-matched items).

Operator instances can have zero or more input and output *ports*. Each output port generates a uniquely named *stream*, which is a sequence of tuples. Connecting an output port to the input of an operator establishes a *stream connection*. Each stream has a *tuple type* associated with it, which can be considered as the schema of the stream. Tuples consist of a list of named and typed *attributes*. Stream connections implement ordered delivery.

DOI: 10.1002/spe

Operators are often implemented in general purpose languages, using an event-driven interface, by reacting to tuples arriving on operator input ports. Processing of a tuple often involves updating some operator-local state and producing result tuples that are sent out to the output ports. A major example of local state that is updated by an operator is *windows*.

Operator instances in SPL can be configured via a few *clauses*. Here, we list the clauses that are highly relevant for our work.

- *Windows* are specified on a per-input port basis and describe how tuples received from the port in question are stored.
- *Parameters* are used to configure operators. Among many other things, such configurations may also relate to windows. For instance, in SPL, the partitioning attributes corresponding to a partitioned window are specified via a parameter.
- *Outputs* are used to specify how output tuples are to be constructed from the input ones. An important concept associated with outputs is *output functions*. Such functions can be used to specify operations to be performed on windows (such as computing the maximum value of an attribute across a window).

We illustrate the use of these clauses in more detail when we formally introduce windowing concepts in the next few sections.

2.2. An example application in SPL

Listing 1 gives the source code for a very simple stream processing application in SPL, with its visual representation depicted in Figure 1.

Listing 1. `SensorQuery`: A simple application in SPL.

```
composite SensorQuery {
  type
    Location = tuple<float32 x, float32 y>;
    Sensor = tuple<uint64 sid, float64 value, Location sloc>;
    Query = tuple<uint64 qid, Location qloc, float64 radius>;
    Result = Sensor, Query, tuple<float32 distance>;
  graph
    stream<Sensor> Sensors = SensorSource() {}
    stream<Query> Queries = QuerySource() {}
    stream<Result> Results = Join(Sensors as S; Queries as Q) {
      window Sensors : sliding, time(10.0);
             Queries : sliding, count(0);
      param match : distance(S.sloc, Q.qloc) <= Q.radius;
      output Results : distance = distance(S.sloc, Q.qloc);
    }
    () as Sink = TCPSink(Results) {
      param
         role : client;
         address : "192.168.0.10";
         port : 40000;
    }
}
```

The application is named `SensorQuery` and contains two *source operators*. Source operators do not have any input ports. One of the sources is an instance of a `SensorSource` operator and generates a stream of sensor readings, namely, the `Sensors` stream. The other source is an instance of a `QuerySource` operator and generates a stream of queries defined over the sensor readings, namely, the `Queries` stream. The `Sensors` and `Queries` streams are connected to the two input ports of a `Join` operator instance. This particular instance of the `Join` operator is configured such that it has a sliding window defined on the `Sensors` stream (its first input port). This window is used to keep the last 10 s worth of tuples. The operator also defines an empty window on the `Queries` stream. This is an example of a *one-sided join*. For each query tuple it receives, the join operator will try to match the query tuple against the sensor tuples stored in the window associated with the `Sensors` stream, based on a match condition that checks if the sensor reading is within the region of interest specified by the query. Effectively, the join operator is finding the sensor readings

DOI: 10.1002/spe

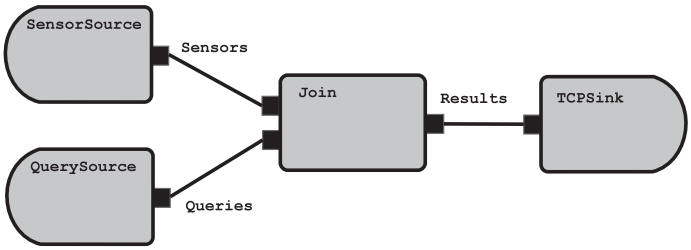

Figure 1. Data flow graph for the `SensorQuery` application.

from the last 10 s that match the current query. The matches are produced by the `Results` stream (the only output port of the join) and are fed into a *sink operator*. Sink operators do not have any output ports. In this particular case, the sink is an instance of a `TCPSink` operator, which writes the results on a TCP socket.

This example illustrates the role of windows in stream processing, as well as how operator composition and parameterization works in SPL. Many generic stream processing operators need to work with windows. In this application, we have seen a join operator employing windows. Another common example is relational aggregations. However, the need for windowing is not limited to relational operators. For instance, a de-duplication operator in a telecommunication application needs windows to define how much history it should maintain, or a sink operator used to perform batched writes to a storage subsystem needs windows to define the batch size. Similarly, a correlation operator that detects similarity across multiple streams need to specify how much of the stream history is to be kept for the analysis.

2.3. Runtime support

A distributed stream processing middleware, such as System S, executes data flow graphs by partitioning them into basic units called *processing elements*. Each processing element contains a subgraph and can run on a different host. Stream connections between operators that are contained within the same processing element are often implemented via efficient mechanisms such as function calls, whereas stream connections that cross processing element boundaries rely on a transport subsystem such as TCP/IP. The streaming runtime provides services such as scheduling, placement, security, and fault-tolerance. Further details can be found elsewhere [5]. In this paper, we focus on the windowing runtime, which is local to each operator in a processing element.

3. WINDOWING CONCEPTS

In this section, we introduce windowing concepts, illustrate them with SPL syntax, and most importantly, give detailed semantics for different window configurations.

3.1. Window types

There are two main types of windows employed in stream processing systems. These are *tumbling* windows and *sliding* windows [1, 2, 15]. Both types of windows store tuples in the order they are received but differ in how they handle *evictions* and *triggers*.

3.1.1. Tumbling windows. A tumbling window stores tuples until the window is *full*. Once the window is full, it is ready for processing. The processing performed on the window is specific to the operator at hand. After the window processing is complete, all the tuples in the window are evicted, effectively emptying the window. We name the policy that defines when a tumbling window is full as the *eviction policy*. We will look at various different eviction policies shortly, but the most basic policy is the count-based one, where a pre-defined maximum size determines when the window is full.

DOI: 10.1002/spe

Listing 2. Use case for tumbling windows

```
composite SampleAggregation(
    input stream<rstring tid, int64 value> In,
    output stream<rstring tid, int64 totalValue> Out) {
  graph
    stream<Out> Out = Aggregate(In) {
      window In : tumbling, count(100);
      param groupBy : In.tid;
      output Out : totalValue = Sum(In.value);
    }
}
```

Listing 2 illustrates the syntax of tumbling windows in SPL using an aggregation scenario. In this example, tuples are stored in a tumbling window of size 100. Once the window is full, the tuples are grouped based on their transaction id (`tid`) attribute and for each unique tid value in the window, an output tuple with the sum of the `value` attributes of tuples sharing that `tid` is generated.

3.1.2. Sliding windows. A sliding window continuously maintains the most recent tuples. When a new tuple is inserted into a sliding window, zero or more of the oldest tuples that are currently in the window are evicted. The evictions happen when the window is already full upon a tuple insertion. Similar to tumbling windows, the *eviction policy* determines when the window is full. Unlike tumbling windows, only the oldest tuples are evicted to make up room for a new tuple.

Sliding windows also have a *trigger policy* associated with them, which determines when the window is ready for processing. Unlike tumbling windows, which are processed when they get full, sliding windows are processed on the basis of their trigger policy. Different kinds of eviction and trigger policies can be combined, as we will discuss shortly. A simple combination is a sliding window with a count-based eviction policy and a count-based trigger policy, which we illustrate with an example in the succeeding text.

Listing 3. Use case for sliding windows

```
composite SampleAggregation(
    input stream<rstring tid, int64 value> In,
    output stream<rstring tid, int64 totalValue> Out) {
  graph
    stream<Out> Out = Aggregate(In) {
      window In : sliding, count(100), count(10);
      param groupBy : In.tid;
      output Out : totalValue = Sum(In.value);
    }
}
```

Listing 3 illustrates the syntax of sliding windows in SPL using an aggregation scenario. In this example, tuples are stored in a sliding window that has a count-based eviction policy of size 100. What this means is that, at any time, the window keeps the last 100 tuples. The window also has a count-based trigger policy of 10, which means that the window is processed every 10 tuples. In this particular example, after every 10 tuples, the last 100 tuples are grouped on the basis of their tid attribute and for each unique tid value, an output tuple with the summation of the `value` attributes of tuples sharing that `tid` is generated.

One subtle issue with sliding windows is whether their processing should be triggered before a full window has been seen or not. This may happen during initial phases of the execution. In the example given in Listing 3, when the application is first launched, after receiving the first 10 tuples, the trigger policy will initiate window processing, yet, the window does not have 100 tuples in it. Whether this is acceptable or not depends on the application semantics. The windowing library we describe in this paper exposes this choice to operator developers. For instance, the `Aggregate` operator supports an `aggregatePartialWindows` parameter that can be used to change the behavior with respect to partial window processing.

DOI: 10.1002/spe

The eviction and trigger policies for sliding windows are sometimes referred to as the *range* and *slide* of the window, respectively, in the literature [16]. However, this terminology does not generalize to all combinations of policies that we introduce in this paper.

3.1.3. Notation. We now introduce lightweight notation to represent windows, which we later use to formally define the semantics of different eviction and trigger policy combinations.

We denote a window as $W = \{t_i \mid i \in [0..|W|-1]\}$, where $|W|$ denotes the window size in terms of the number of tuples, and t_i is the ith tuple in the window. t_0 represents the most recent tuple in the window, whereas t_l represents the oldest tuple in the window ($l = |W|-1$). We use t_c to denote the current tuple that is received and to be inserted into the window but not yet inside the window. For a tuple t, we use $\tau(t)$ to denote its arrival timestamp. The arrival time is defined as the time of arrival into the window and is used for time-based windowing policies. Windows that are defined on timestamps attributes carried within tuples are handled via attribute-delta windowing policies.

3.2. Partitioning

Before we describe the details of eviction and trigger policies, we introduce an important variety of windows, called *partitioned* windows. Partitioned windows are central in supporting multiplexed processing, which we examine first.

3.2.1. Multiplexed processing. In many stream processing applications, streams contain multiple logical substreams in them. It is often the case that the number of unique substreams multiplexed in a stream is not known at application development time or changes at runtime. As one example, consider a stream containing market feed data. In this case, a substream corresponds to the series of tuples that share their value for the ticker symbol. As another example, consider a stream containing network protocol information. In this case, a substream corresponds to the series of tuples that share their IP address attribute value.

Given a multiplexed flow, a common need is to perform a computation on each substream, independently. A typical example is computing the volume weighted average price (VWAP) for each stock ticker in a financial trading application. In other words, the VWAP value is computed over each substream independent of the others.

We call the attributes that partition a stream into a number of substreams *partition-by* attributes. In the VWAP example, the partition-by attribute is the stock ticker attribute. Once partition-by attributes are defined, a stream can be processed in a multiplexed fashion using stateless operators or stateful operators that support multiplexed operations.

A stateful operator supports multiplexed processing iff (i) it keeps independent state for each substream and (ii) for each tuple it receives, it only reads and/or updates the state associated with the substream the tuple it belongs to. One of the fundamental forms of local state in stream processing is windows. As such, to support multiplexed processing, windows must support partitioning as well. For instance, VWAP computation requires computing averages over a window of data. In fact, that is the only stateful piece of computation involved.

Multiplexed processing is also important because of the role it plays in parallelization [13]. Data parallelism in the form of splitting data across multiple copies of an operator (or a series of operators) and merging back the results can be effectively applied when operators are either stateless or stateful with support for multiplexed processing. In the latter case, the splitting should be performed by making sure that tuples with the same partition-by attribute value are always assigned to the same copy of the replicated operator.

3.2.2. Partitioned windows. A *partitioned window* contains subwindows, each corresponding to a different substream identified by the value of its partition-by attributes. It is as if there is one window for each substream, operating completely independent of the windows of other substreams.

Listing 4 extends the example from Listing 3 with a partitioned window. The partitioned nature of the window is indicated via a `partitioned` keyword, which in SPL requires a `partitionBy` parameter to be specified. In this particular example, the partitioning attribute is `state`. As a result,

DOI: 10.1002/spe

Listing 4. Use case for partitioned sliding windows

```
composite SampleAggregation(
    input stream<rstring tid, rstring state, int64 value> In,
    output stream<rstring tid, rstring state, int64 totalValue> Out) {
  graph
    stream<Out> Out = Aggregate(In) {
      window In : sliding, count(100), count(10),
                  partitioned;
      param groupBy : In.tid; partitionBy : In.state;
      output Out : totalValue = Sum(In.value);
    }
}
```

the window will keep the last 100 tuples for all unique values of `state` seen so far. When 10 tuples are received for a given subwindow, then processing is triggered for that subwindow. Concretely, the tuples in the subwindow will be grouped on the basis of their tid attribute, and for each unique tid value, an output tuple with the summation of the `value` attributes of tuples sharing that `tid` is generated. The key difference from the previous example is that, the trigger policy and the eviction policy apply to each subwindow separately, and the processing is performed on each subwindow independently.

Partitioned windows apply equally to tumbling and sliding windows. However, there is a fundamental difference in terms of how subwindows are created and removed. For a tumbling window, when a subwindow gets full, it can be discarded after it is processed by the operator. On the other hand, for a sliding window, a subwindow may never be removed once it is created, as it may never be empty. Listing 4 is an example where a subwindow exists forever, once it is created. When the total number of substreams is bounded by a small number, this is often not an issue. But when that bound is a large number, the window can run out of memory. An example is a network monitoring application, where the pair of IP addresses form the partitioning attribute. We handle this problem via *partition eviction* policies, which we will discuss later in the paper.

3.2.3. Notation. We now extend our windowing notation to partitioned windows.

We denote a partitioned window as $W^a = \{W^{a=v} | v \in \{t.a \mid t \text{ is a tuple in } W^a\}\}$, where a denotes the partition-by attribute. We use $|W^a|$ to denote the number of subwindows in W^a, and $W^{a=v}$ denotes a subwindow that is associated with the partition-by attribute value of v. It is defined as $W^{a=v} = \left\{t_i^{a=v} | \, i \in [0..|W^{a=v}|)\right\}$. Here, $t_0^{a=v}$ represents the most recent tuple in the subwindow $W^{a=v}$, whereas $t_l^{a=v}$ represents the oldest tuple. We use $t_c^{a=v}$ to denote the current tuple that is received and to be inserted into the subwindow but not yet inside it.

3.3. Window policies

In this section, we introduce different windowing policies and specify operational semantics for window configurations that result from application of these policies on tumbling and sliding windows. These policies are divided into four, namely, *count*, *attribute-delta*, *time*, and *punctuation*-based policies. The first three can appear as both eviction and trigger policies, whereas the last one can only appear as an eviction policy.

There are three fundamental operations performed as part of maintaining tuples in a window. These are *insertion*, *eviction*, and *trigger*. Insertion represents the addition of the current tuple $\left(t_c^{a=v}\right)$ into its respective subwindow $(W^{a=v})$ as the most recent tuple. Eviction represents removal of one or more of the oldest tuples from the window. Trigger represents calling out the operator logic for processing the window. Note that for tumbling windows, the eviction and trigger events overlap.

Insertion, eviction, and trigger events are performed in different orders and in different ways depending on the window type and the eviction and trigger policies being applied. We provide Tables I and II as reference for the order of these events for tumbling and sliding windows, respectively. The $\rightarrow$ notation is used to indicate what happens before a relationship between events, whereas the | notation is used to indicate that the two events happen independently.

DOI: 10.1002/spe

Table I. Order of events for tumbling windows.

Eviction policy	Order of execution
Count-based	First, insert tuple into window, then perform eviction.
Delta-based	First, perform eviction, then insert tuple into window.
Time-based	Perform evictions independently of tuple insertions.
Punctuation-based	Perform eviction when a punctuation is received.

Table II. Order of events for sliding windows.

Eviction \ trigger	Count-based	Delta-based	Time-based
Count-based	evict → insert → trigger	trigger → evict → insert	evict → insert | trigger
Delta-based	evict → insert → trigger	trigger → evict → insert	evict → insert | trigger
Time-based	insert → trigger | evict	trigger → insert | evict	insert | evict | trigger

3.3.1. Count. A count-based policy is characterized by a size in terms of number of tuples, denoted by n.

As an *eviction policy for tumbling windows*, the count-based policy defines the maximum number of tuples in a subwindow. On the arrival of a new tuple $\left(t_c^{a=v}\right)$, this tuple is inserted into its associated subwindow (becomes $t_0^{a=v}$ in $W^{a=v}$). If the number of tuples in the subwindow is equal to the size specified by the count-based eviction policy ($|W^{a=v}| = n$), then the window is flushed (all tuples are evicted after window processing is performed).

As an *eviction policy for sliding windows*, the count-based policy defines the maximum number of tuples to be kept in a subwindow as it *slides*. On the arrival of a new tuple $\left(t_c^{a=v}\right)$, if the number of existing tuples in its associated subwindow is already equal to the size specified by the eviction policy ($|W^{a=v}| = n$), then the oldest tuple in the window $\left(t_l^{a=v}\right)$ is evicted. Once the eviction is complete (if needed), the newly arrived tuple is inserted into the subwindow.

As a *trigger policy for sliding windows*, the count-based policy defines the number of tuples that must be received by a subwindow until an operator's internal processing is triggered for that subwindow. Before the operator logic runs, evictions are performed (if needed) first and the new tuple is inserted next. The particular eviction action to be performed depends on the eviction policy (which may not be count based). At this point, the trigger counter is reset for the subwindow at hand, restarting the cycle.

For sliding windows with count-based trigger policies, the order of operation is given as follows for a newly arrived tuple: eviction, insertion, and trigger. The only exception is when a time-based eviction policy is involved, in which case eviction is independent of the insertion and trigger, as we will discuss later.

3.3.2. Attribute-delta. An attribute-delta policy is characterized by a *delta threshold value*, denoted by δ, and a *delta attribute*, denoted by d. The delta attribute refers to an attribute in the tuple type corresponding to the input port associated with the window in question. The values for the delta attribute should be non-decreasing. A typical example is a timestamp attribute, which represents the time associated with the tuple in question as assigned by an external or internal source but different than the wall-clock time of the tuple's arrival to the window.

As an *eviction policy for tumbling windows*, the attribute-delta-based policy defines how far the oldest and the newest tuples stored in a subwindow can deviate from each other in terms of their delta threshold value. This is called the *delta invariant*: $\forall_{W^{a=v} \in W^a}, t_0^{a=v}.d - t_l^{a=v}.d \leqslant \delta$. A subwindow is considered full when the next tuple to be admitted would break the delta invariant.

On the arrival of a new tuple $\left(t_c^{a=v}\right)$, if the difference between the new tuple's value for the delta attribute and that of the oldest tuple in the window is larger than the delta threshold value, that is $t_c^{a=v}.d - t_l^{a=v}.d > \delta$, then the subwindow is processed and the tuples are evicted. Later, the newly arrived tuple is inserted into an empty subwindow.

Note that the order of processing is different for attribute-delta-based tumbling windows, compared with the count-based ones, as the eviction is performed before insertion. This is because

DOI: 10.1002/spe

the newly arrived tuple that breaks the attribute-delta invariant is not part of the closed window. As a result, the window needs to be processed first in its current state, before the new tuple is inserted. This has an important implication, that is, a subwindow in a partitioned tumbling window never becomes empty. This is because, after the subwindow is processed and its contents evicted, the current tuple is inserted, leaving $W^{a=v} = \{t_0^{a=v}\}$. We address this later via partition eviction policies.

As an *eviction policy for sliding windows*, the attribute-delta-based policy defines the range of tuples maintained in the subwindow as it slides, again using the delta invariant. On the arrival of a new tuple $(t_c^{a=v})$, the existing tuples in its associated subwindow ($W^{a=v}$) for which the difference between the newly received tuple's value for the delta attribute and that of the existing tuple's is greater than the delta threshold values are evicted. This set of tuples is formally represented as $\{t \mid t \in W^{a=v} \wedge t_c^{a=v}.d - t.d > \delta\}$. The newly received tuple is inserted into the subwindow after the evictions take place (if any).

As a *trigger policy for sliding windows*, the attribute-delta-based policy defines when the operator internal logic is executed. Hence, on the arrival of a new tuple, if the difference between the new tuple's value for the delta attribute and that of the last tuple that triggered the processing for the subwindow is greater than the delta threshold, then the subwindow is processed again. Note that this requires keeping around the last tuple that triggered the processing for the subwindow, even if that tuple is already evicted. If any tuples are subject to eviction, these evictions are performed *after* the window processing is complete. Again, the specifics of the eviction to be performed depends on the eviction policy being used, which is not necessarily attribute-delta based.

For sliding windows with delta-based trigger policies, the order of operation is given as follows for a newly arrived tuple: trigger, eviction, insertion. The only exception is when a time-based eviction policy is involved, in which case eviction is independent of insertion and trigger. It is worth noting that, yet again, the order is different than the one for count-based windows.

To see that this is the expected semantics, one can compare the operation of a sliding window with a count-based eviction policy with size n (say 2) and a count-based trigger policy with size m (say 1), to that of a sliding window with a attribute-delta-based eviction policy with threshold $n - 1$ (say 1) and attribute-delta-based trigger policy with thresholds $m - 1$ (say 0), assuming that the delta attribute is a monotonically increasing integer. Assuming partial windows during initialization are not processed, in the former case, the second tuple will trigger the window processing for the first time. Although the window is being processed, the second tuple would already be inside the window (because insertion comes before trigger), thus, the window will contain the first two tuples. In the latter case, the third tuple will trigger the window processing for the first time. Although the window is being processed, the third tuple would not be inside the window (because insertion comes after trigger), thus, the window will still contain the first two tuples. Figure 2 illustrates this equivalence in terms of the processed windows. This equivalence is important to achieve consistent semantics. In this particular case, we can emulate a window with configuration `sliding, count(n), count(m)` using a window with configuration `sliding, delta(id, n-1), delta(id, m-1)`, assuming that id is a monotonically increasing integer attribute with step size 1. The latter configuration showcases the syntax used by SPL for representing attribute-delta-based windows, where the first parameter to the `delta` function is the delta attribute name and the second one is the delta threshold. In general, attribute-delta-based windows cannot be emulated by count-based ones, but when the delta attribute is monotonically increasing discrete type, the expectation is that the count-based policy and delta-based policy result in processing the same set of windows.

`sliding, count(2), count(1)`	`sliding, delta(id, 1), delta(id, 0)`
tuple 0	tuple 0
tuple 1 → trigger processing, window = (1, 0)	tuple 1
tuple 2 → trigger processing, window = (2, 1)	tuple 2 → trigger processing, window = (1, 0)
tuple 3 → trigger processing, window = (3, 2)	tuple 3 → trigger processing, window = (2, 1)
...	tuple 4 → trigger processing, window = (3, 2)
	...

Figure 2. Equivalence between two different window configurations.

DOI: 10.1002/spe

3.3.3. Time. A time-based policy is characterized by a period in wall-clock time denoted by p.

As an *eviction policy for tumbling windows*, the time-based policy defines the period required to consider that a window is full. When the wall-clock time elapsed because the last time the subwindow was flushed exceeds the period specified by the eviction policy (p), the operator logic is invoked and the subwindow is again flushed. Differently from other policies, the processing of the window and the following eviction event is performed independently of tuple arrivals.

As an *eviction policy for sliding windows*, the time-based policy defines how long a tuple can stay in its subwindow. Tuples that have been in the window longer than the period specified by the policy are evicted. That is, the *age invariant* is maintained: $\forall_{W^{a=v} \in W^a}, \forall_{t \in W^{a=v}}, \tau_c - \tau(t) \leqslant p$ where τ_c represents the current time. Again, tuple evictions take place independently of tuple insertions.

As a *trigger policy for sliding windows*, the time-based policy defines the period between successive processing of a subwindow. The arrival of a new tuple is *not* necessary for the operator to process the tuples that have been stored so far.

3.3.4. Punctuation. Window punctuations represent boundaries within a stream, which separate the flow of tuples into a series of windows. Punctuation-based eviction policies only apply to tumbling windows. A tumbling window with a punctuation-based eviction policy stores tuples until it sees a punctuation, at which time the window is processed and then the tuples evicted. Window punctuations enable propagating window boundaries across operators. We now look at an example to illustrate the use of window punctuations.

Consider the example from Figure 1 where a `Join` operator was used to find the list of sensor readings matching a stream of queries. For each query, there could be zero or more sensor readings produced. Let us assume that we would like to compute the minimum distance from the matching sensor readings to the query, say using an `Aggregate` operator. The particular problem we run into is that the aggregation needs to be performed over a window, where the boundaries of the window are defined by the upstream operator based on the content of its input data and are not known at development time. Window punctuations address this problem. In particular, the join operator produces a window punctuation after each set of matches it generates, and the downstream aggregate operator can specify a punctuation-based window to aggregate the sensor readings that correspond to the same query, in order to compute the minimum distance. Listing 5 gives the SPL code for this extension. However, for this kind of logic to work, we need to define the behavior of operators with respect to window punctuations, as well as how these punctuations propagate across operators.

Listing 5. An extension to `SensorQuery` application to compute minimum sensor distance.

```
// continued from Listing 1

stream<Query, tuple<float32 minDistance>> MinDists = Aggregate(Result) {
  window Result : tumbling, punct();
  output MinDists : minDistance = Min(Result.distance);
}
```

3.4. Punctuation semantics

Knowing the behavior of operators with respect to window punctuations enables a stream processing language compiler to perform compile-time checks to detect semantic violations. For instance, this knowledge can be used to ensure that a tumbling window with a punctuation-based eviction policy is indeed connected to a stream that contains punctuations. If not, this will obviously lead to infinite memory growth at runtime.

We say that a stream is a *punctuated* stream if it carries punctuations generated by one and only one operator. Merging two punctuated streams arbitrarily will result in a stream that is not punctuated, as the window boundaries will be garbled. A data flow graph *respects punctuation semantics* if and only if all ports that are *expecting* window punctuations are connected to a punctuated stream.

DOI: 10.1002/spe

An example of a port that expects window punctuations is one that has a window defined on it with a punctuation-based policy.

We now define punctuation *modes* associated with input and output ports of operators to provide concrete rules for verifying punctuation semantics. These modes are used by the compiler to perform semantic checks. As we have discussed in Section 2, user-defined operators are often developed in general purpose programming languages such as C++/Java. Thus, we require the operator developer to declare the punctuation modes of their operators as part of the operator development. This way, the application developers are shielded from potential errors in composition that will violate punctuation semantics. We describe rules for checking punctuation semantics as well.

An operator input port may have one of the following window punctuation modes:

- *Oblivious*: The input port does not require window punctuations to work correctly.
- *Expecting*: The input port does require window punctuations to work properly.
- *Window-bound*: The input port is *Expecting* when configured with a window that uses punctuation-based eviction policy, and *Oblivious* otherwise. Any given instance of the operator would either be *Oblivious* or *Expecting* on this port.

An operator output port may have one of the following window punctuation modes:

- *Free*: The output port does not carry any window punctuations.
- *Generating*: The output port does generate its own window punctuations.
- *Preserving*: The output port preserves the window punctuations it receives from one of the input ports. In this case, the input port from which the punctuations are being preserved is also specified.

Given these definitions, we can more formally define a punctuated stream. A stream is *punctuated* if and only if one of the following two conditions hold:

- The stream is generated by an output port that has a window punctuation mode of *Generating*.
- The stream is generated by an output port that as a window punctuation mode of *Preserving* and the input port from which the window punctuations are being preserved is connected to a single *punctuated* stream.

There are two important things to note here. First, the definition is recursive. A punctuated stream can go through a number of preserving operators and eventually reach to a punctuation expecting input port. For instance, in our SPL example from Listings 1 and 5, the output from the join operator instance is a punctuated stream, because the `Join` operator's output port is defined as *Generating*. We could have applied some transformation to this output before we feed it to the instance of the `Aggregate` operator that has a punctuation-based window on its input port. As long as the operator that does the transformation preserves the window punctuations in the join output, the punctuation semantics are still maintained. Second, the definition of the *punctuated* stream implies that connecting more than one stream to an input port from which punctuations are being forwarded results in an output stream that is *not* punctuated. This is because, as we have mentioned earlier, because of losing punctuation semantics as a result of arbitrarily intermixing window boundaries marked in one stream with tuples from another one.

Given our definition of a punctuated stream, the compiler can easily compute which streams are punctuated and verify that all punctuation expecting input ports are connected to a single punctuated stream.

3.5. Partition eviction

We have seen that all variations of partitioned sliding windows as well as partitioned tumbling windows with attribute-delta-based eviction policies result in creation of subwindows that are never removed. We address this problem via *partition eviction*.

DOI: 10.1002/spe

Partition eviction has two pieces to it. The first is *partition eviction policy*, which determines the conditions under which subwindows should be evicted. The second is *partition selection policy*, which determines the subwindows to be evicted.

3.5.1. Partition eviction policies. We define three common partition eviction policies, namely, *partition count*, *tuple count*, and *partition age*. Partition eviction policies can be used with both tumbling and sliding windows.

The partition count policy specifies the maximum number of subwindows that can be present in a window, say r. When the number of subwindows exceeds this threshold ($|W^a| > r$), one is evicted to make space.

The tuple count policy specifies the maximum number of tuples that can be kept in a window across all subwindows. This is reflective of the entire window size and can be more directly related to memory usage. When the number of tuples in the window exceeds the threshold $\left(\sum_v |W^{a=v}| > r\right)$, then one or more subwindows are removed until there is no need for partition eviction.

The partition age policy specifies the maximum amount of wall-clock time a subwindow can remain in a window without any updates. In other words, we define the *age* of the subwindow as the amount of time passed since the last update that modified the subwindow. When the age of a subwindow exceeds the threshold, it is removed.

Unlike the partition age policy, the partition count and tuple count policies rely on a partition selection policy to determine the partition to remove.

3.5.2. Partition selection policies. Partition selection policies can vary in practice and can even be specific to the operator at hand (*user-defined*). Here, we list a few common ones, such as the *least recently updated partition* (LRU), the *least frequently updated partition*, and the *oldest partition*. Our windowing library implements the LRU policy by default and provides a user-defined option for operator-specific policies.

Listing 6. Use case for partition eviction policy

```
composite SampleAggregation(
    input stream<timestamp ts, rstring srcIp, rstring dstIp, uint64 traffic> In,
    output stream<timestamp ts, rstring srcIp, rstring dstIp, uint64 totalTraffic> Out) {
  graph
    stream<Out> Out = Aggregate(In) {
      window In : sliding, delta(ts, 100.0), delta(ts, 10.0),
                  partitioned, partitionAge(hour());
      param partitionBy: In.srcIp, In.dstIp;
      output Out : totalTraffic = Sum(In.traffic);
    }
}
```

Listing 6 shows an example use case for partition eviction policy in SPL. In this example, we have an aggregation to be performed on a stream of tuples that contain source and destination IP addresses. The goal is to perform a summary over a sliding window. The summary to be computed is the total amount of traffic for each unique IP address pair. The `partitionAge` clause is used to specify that a subwindow that has not been updated for more than an hour should be removed. This addresses the memory growth problem that would have been present otherwise.

4. RUNTIME LIBRARY

In this section, we describe the design and implementation of our runtime library. This library is used by user-defined operators to implement windowing logic consistent with the semantics we have described in Section 3.

4.1. Design

The design of the windowing library brings several challenges. First, the library should be generic enough to work with arbitrary stream types and should support flexible back-ends for tuple storage.

DOI: 10.1002/spe

Our design addresses this challenge by modeling a window as a templatized *container* of tuples, with support for customizing the tuple types and data structures used to store them. Second, the library should separate the windowing semantics from the operator-specific window processing logic. We address this challenge by using a *policy-based* window configuration and an *event-based* interface for performing processing on the window contents. Last but not the least, the library should support extensions that allow the operator implementer to decide which tuples are kept in the window. This enables the creation of advanced synopsis structures that use the familiar windowing semantics we have outlined earlier. To address this, we introduce the concept of *window summarizers*, which leave the decision of what tuples are to be stored in memory to a user-defined routine, while keeping the window semantics intact, for tumbling windows.

4.1.1. Templatization. We provide two templatized container types: *TumblingWindow* and *SlidingWindow*. They both derive from a common *Window* container. The reason tumbling and sliding windows are differentiated is because they provide a non-overlapping set of events, as we will discuss shortly.

Window containers are templatized on four dimensions. The first is the tuple type they hold. The second one is the type of the partition-by attributes for the window. This template parameter has a default and does not need to be specified for windows that are not partitioned. The remaining two template parameters are used to control the data structures used for the subwindow and the map that manages the set of subwindows, respectively. We use a double-ended queue for the former and a hash table for the latter, by default. These template parameters are present for future extensibility of the library, such as providing a disk-based storage.

4.1.2. Policy specification. The key design point of the library is its policy-based design and event-based interface. Concretely, to construct a window, the container object is initialized with a set of policies. These include the eviction, partition eviction, and partition selection policies for tumbling windows, where the last two are optional. For sliding windows, a mandatory trigger policy has to be specified as well. Irrespective of what these policies are, the processing logic of the operator stays the same and has to only deal with reacting to *window events*. For instance, an operator that supports a partitioned sliding window with an attribute delta-based eviction policy and a time-based trigger policy as well as a non-partitioned window with a count-based eviction and trigger policies is implemented the same way for both configurations, except the initialization of the window container, where the windowing policies are specified.

4.1.3. Window events. Window events are fired as a result of changes in the window that are relevant from the perspective of the operator implementation. Window events are created in response to insertion of tuples into the window, as well as due to passage of time (as time-based policies do not depend on tuple insertions).

All window events are grouped together in a *WindowEvent* interface as a set of event handlers. Operators that wish to have windowed logic implement this interface and override event handlers of interest. They also register to receive the events they are interested in. The registration enables the runtime to avoid the overhead of tracking and delivering events that are not going to be used by the operator implementation.

We define the following events:

- *Insertion events*. These events are fired before and after a tuple insertion. They provide a handle to the tuple participating in the insertion, the value of the partition-by attribute corresponding to the subwindow we are inserting into (this parameter can be ignored for non-partitioned windows) and a handle to the window object itself (because multiple windows can be used by an operator that has multiple input ports). These events are available for both tumbling and sliding windows.
- *Flush events*. These events are specific to tumbling windows, and they fire before and after a tumbling window is flushed. They provide the partition-by attribute value for the subwindow that is being flushed and a handle to the window.

DOI: 10.1002/spe

- *Eviction events*. These events are specific to sliding windows, and they fire before and after a tuple is evicted from the window. They provide the same set of parameters as the insertion events, that is, the tuple, the partition-by attribute value, and the window handle.
- *Trigger event*. This event is specific to the sliding windows and is fired when the window is triggered for processing. It provides the partition-by attribute value for the subwindow that was triggered and a handle to the window.
- *Initial full event*. This event is specific to the sliding windows and is fired when the window is full for the first time. It provides the partition-by attribute value for the subwindow that became full and a handle to the window. Recall from Section 3.1.2 that this event is required to implement the option of not executing the window processing for partially full windows during initialization.
- *Partition eviction event*. This event is fired when subwindows are being evicted from the window. It provides an iterator that can be used to inspect the subwindows before they are evicted. It is available for both tumbling and sliding windows.
- *Partition selection event*. This event is fired when one or more subwindows are to be selected for eviction but only when a user-defined partition selection policy is used. It provides an iterator that can be used to inspect the subwindows and mark some of them for eviction. It is available for both tumbling and sliding windows.

4.1.4. Window access. The windowing library also provides access APIs so that the contents of the window can be inspected as part of the event handler processing. This includes iterating over the subwindows and their associated partition-by values, as well as iterating over the tuples within subwindows. These APIs are safe to use within event handlers. However, to access these APIs from outside of event handlers, the developers need to acquire the window lock, as the potential presence of time-based events may result in concurrent modifications to the window. The acquiring of the window lock is managed through a scoped object, which makes sure that the locking cost is not incurred when the particular instance of the window object at hand does not use any time-based policies.

Scoped window locks come in handy when the window contents are to be processed as a result of some event other than the events generated by the window itself. An example is operators that have more than one input port and thus more than one window (one per input port). Such windows can have independent configurations and one of them may need to be processed when an event fires on another. An example for this is the join operator, which is further detailed in Section 6.1.2.

4.1.5. Putting it all together. The steps involved in using the windowing library from within a user-defined operator can be summarized as follows. The window object, which is a container, is templatized with the relevant types and is instantiated with the policies that are borrowed from the invocation of the operator instance at hand. Relevant window events are registered and corresponding handlers are implemented. As tuples are received, they are inserted into the window, which results in the firing of the various event handlers. The operator performs its logic as part of the event handlers. This is often achieved by using the window access APIs to inspect the contents of the window and carry out operator-specific processing.

As an example, consider an `Aggregate` operator using a tumbling window to implement a computation that involves averaging the tuples in the window. The operator can register for the before window flush event and compute the average by going over the tuples in the window before the tuples are evicted.

This example also brings up an important issue: There are various computations that do not necessarily require keeping all the tuples in the window. A good example of this is incremental aggregates, such as sum, average, and standard deviation. Furthermore, there are other computations that may not require the full contents of the window but only a sample of them, such as aggregations that provide probabilistic guarantees on their accuracy and the amount of space they use [17].

There are two problems with physically storing all the tuples in the window, irrespective of whether the processing needs them all at once or not. The first one is an issue of memory usage. For large windows, we will run out of memory, even though an incremental computation would not.

DOI: 10.1002/spe

The second one is an issue of computation. Without incremental computation, we end up with periods of little per-tuple processing (tuple insertion) followed by a period where a single tuple takes a long time to process, as it triggers the window processing. This results in stop-and-go behavior in an asynchronous stream processing system.

We address these issues with *window summarizers*.

4.1.6. Window summarizers. A window summarizer is a piece of user-defined logic and state that determines what information needs to be kept in a subwindow. When a tumbling window is configured with a window summarizer, it only keeps enough data to implement its policies and does not store the individual tuples in the subwindows. Instead, it creates a window summarizer for each subwindow and delivers the tuples that arrive at the window to the summarizers. It is up to the summarizer to determine the amount of state to keep. The windowing library provides APIs to access window summarizers associated with subwindows, which can be used to retrieve the summarizer from within window events (such as a window flush).

All window summarizers implement an interface called *WindowSummarizer* and override the three following basic event handlers as follows:

- *Open window event*. This event is fired when a new subwindow is created and thus immediately after a new window summarizer is created. This event provides the partition-by attribute value that corresponds to the subwindow.
- *Tuple insertion event*. This event is fired when a tuple is logically inserted into the window. The window container will not keep this tuple physically inside the window. This event provides a handle to the tuple and it is up to the summarizer to decide whether it is to be kept around or not.
- *Close window event*. This event is fired when a subwindow is flushed and thus immediately before an existing window summarizer is destroyed.

The window summarizer events provide complete control over what needs to be stored in a tumbling window. It is possible to implement algorithms that require constant state (such as an incremental aggregation), logarithmic state (such as approximate frequency counting [17]), or complete state (such as Fast Fourier Transform (FFT)). However, keeping a complete state is not a good use case for window summarizers, as that is the default behavior without the use of window summarizers. In general, window summarizers are useful when the state to be kept is sublinear in the size of the window.

4.1.7. An end-to-end example. Listing 7 provides an example use of window summarizers. We assume that this is an operator that uses a partitioned tumbling window to compute an average over an attribute called `value`, where an attribute called `partition` is used as the partition-by attribute. We further assume that the window is a count-based one with size 1,000,000.

The code is given in C++. The first function we see is the operator initializer, which creates the `window_` object (a member variable) with a count-based eviction policy. It also registers for the window flush event, asking the window to deliver this event to an instance of `WindowHandler`, which is called `windowHandler_` (another member variable). Finally, it registers a window summarizer class, named `SummarizerHandler`, with the window.

The second function we see is the tuple process function of the operator. This function is called on the operator whenever a tuple is received. The operator uses this tuple to call the `insert` function on the `window_` object, passing it the tuple, as well as the partition it belongs to. The latter is used to associate the tuple with the right subwindow. Because the window is configured with a summarizer, the insert operation does not necessarily keep the tuple inside the window.

The third function we see is the before window flush handler, which is called on the `windowHandler_` object when the window is ready to be processed. As part of handling this event, we ask the window object to retrieve the summarizer for the subwindow at hand, using the partition-by value provided by the event handler. This is performed using the `getWindowSummarizer` function. Then, we ask the window summarizer to give us the average

DOI: 10.1002/spe

Listing 7. Use of window summarizers for computing averages

```
void Operator::init() {
  window_ = new TumblingWindow(CountBasedPolicy(1000000));
  windowHandler_ = new WindowHandler(this);
  window_->registerOnWindowFlush(windowHandler_);
  window_->registerWindowSummarizer<SummarizerHandler>();
}
void Operator::process(Tuple & tuple, Port const & port) {
  window_->insert(tuple, tuple.get_partition());
}
void WindowHandler::beforeWindowFlush(Partition const & part, Window const & window) {
  SummarizerHandler & handler = (SummarizerHandler &) window.getWindowSummarizer(part);
  float average = handler.getAverage();
  // ... submit a tuple with sum
}
void SummarizerHandler::onOpenWindowEvent(Partition const & part) {
  n_ = 0; // n_ is a member variable keeping the number of tuples seen
  average_ = 0.0; // average_ is a member variable keeping the current average
}
void SummarizerHandler::onTupleInsertion(Tuple const & tuple) {
  float value = tuple.get_value();
  average_ += (value - average_) / (n_ + 1);
  n_++;
}
void SummarizerHandler::getAverage() {
  return average_;
}
```

value. The assumption is that the window summarizer have already computed this value incrementally, as tuples were delivered to it in the past. The operator can then use the average value to create and submit an output tuple, which is not shown in the code listing.

The fourth function we see is the open window event handler for the window summarizer. The summarizer simply initializes a count (`n_`) and an average (`average_`) member variable when the window is opened. The fifth function we see is the tuple insertion event handler for the summarizer, which is used to incrementally update the count and average variables, using a numerically sensitive method for computing the average. Finally, the last function we see is the `getAverage` function, a user-defined function on the summarizer, which is used by the window flush handler to retrieve the average.

4.1.8. Summarizers and sliding windows. For sliding windows, the eviction of a tuple requires updating the summary an operator may be maintaining for the current window and thus has to be delivered to the operator logic. This implies that the sliding window has to keep all the tuples around, rendering window summarizers ineffective for reducing the memory usage even for aggregations that can be incrementally maintained.

This does not mean that operators implementing summarizations on sliding windows cannot be used with very large windows. In many cases, the summarizations to be computed are *associative*. This enables decomposing an operator with a large sliding window into two operators: one with a tumbling window feeding and another one that has a sliding window. The goal is to reduce the window sizes and thus the memory requirement. This is best illustrated with an example. Assume that we have an aggregation operator, computing an average over a sliding window with a time-based eviction policy of 1 h and a trigger policy of 1 min. This operator will generate a summary result over the last 1 h's worth of data every 1 min. Keeping the tuples for the last hour could be prohibitive for high input data rates. This operator can be rewritten into two. The first one will compute the average using a tumbling window with a time-based eviction policy of 1 min, which matches the trigger policy of the original sliding window. The results from this operator can be fed into a second operator that is configured with a sliding window with a count-based eviction policy of 60 and a count-based trigger policy of 1. In effect, the second operator would be aggregating 60 of the 1 min summaries. The net result is the same as the summarization being applied (average in this case) is associative. Yet, this rewritten flow uses only 60 tuples plus 1 min's worth of tuples, as opposed to 1 h's worth of tuples.

DOI: 10.1002/spe

4.2. Implementation

The main challenge involved in the windowing library implementation is the handling of large number of window configurations, especially in the case of sliding windows. We tackle this problem as follows.

First, the partitioned windows are used as the default implementation, where a non-partitioned window is represented as a partitioned window with a single subwindow and a default partition-by attribute value. Implementation of partition eviction policies is piggybacked on insertions, which means age-based partition eviction policies are handled lazily. The only complication with partition evictions is the need for cleaning up eviction and trigger policy specific state when subwindows are removed because of partition eviction. The cleaning of such state is handled by the individual eviction and trigger policy implementations (see succeeding text).

Second, and more importantly, each trigger and eviction policy variation has an implementation of its own, where the eviction policy implementations handle both the insertion and eviction functionality. To implement a sliding window with a combination of eviction and trigger policies, a lookup table that corresponds to Table II is used to decide on the order in which the insertion, eviction, and trigger operations are to be executed. In effect, six policy implementations (three for eviction, three for trigger) are used to support the nine combinations with respect to eviction and trigger policies.

Finally, time-based policies rely on independent threads to implement their functionality. The time-based eviction policies are implemented by maintaining a priority queue on the deadlines of the oldest tuples in subwindows. Windows with time-based eviction policies are subject to becoming empty (if no insertions are performed for a prolonged time), in which case the eviction thread will wait for the future insertions to continue its operation. Time-based triggers are implemented via periodic timers. When the time taken by the operator processing associated with the window trigger is longer than the trigger policy duration, the library will trigger the window in succession to catch up.

5. EXTENSIBILITY

Different stream processing operators have different levels of support for windowing. Not all window configurations are supported by all operators. For instance, a relational join supports sliding windows only, and typically, the trigger policy is always count-based and of size one (tuples are processed as they arrive). A sort operator typically operates on tumbling windows. At the other extreme, an aggregate operator supports all possible windowing configurations. Finally, some operators have no support for windows such as a simple filter.

We support syntactic flexibility via two mechanisms: *operator models* and *code generation*.

5.1. Operator models

An operator model describes syntactic and semantic properties of a reusable stream operator. We require operator developers to specify this model, as part of their operator implementation. With respect to windowing syntax, the operator model defines the window configurations supported by the operator. With respect to semantics, the operator model specifies the window punctuation modes for the input and output ports of the operator.

The operator model is used by the compiler to verify that a particular instance of a stream operator is syntactically correct. Furthermore, the punctuation modes are used to ensure correct punctuation semantics across the graph, as outlined in Section 3.4.

5.2. Code generation

We provide a code generation framework for the development of reusable and generic operators. This framework provides a templating mechanism through which an operator developer can customize the code of an operator instance based on its invocation configuration. A compile-time introspection API is provided for this purpose, which can be used by the operator developer to inspect the configuration of the particular operator instance at hand, and specialize the operator code

DOI: 10.1002/spe

based on that. For instance, the entire window configuration is available through the compile-time introspection API. There are several advantages to this scheme.

First, the compile-time introspection can be used to perform programmatic checks on the correctness of the configurations, especially those checks that are not possible to express using the operator model. An example related to window processing is to check for values of particular policy parameters (such as making sure that a join operator's trigger policy is of size 1) or checking the existence of a parameter that must co-exist with a particular windowing configuration (such as `partitionBy` with a partitioned window).

Second, code generation provides the ability to optimize the performance of the code for the specific operator instance at hand. As an example, a one-sided join is a very common use case in streaming, as we illustrated in Listing 1. It can be optimized by avoiding the insertion of the tuples to the zero-sized window altogether. Of course, such optimizations can also be applied at runtime, at the cost of runtime checks that are executed for each tuple.

Finally, the separation of the syntax from the implementation provides the option not to use our windowing library for the implementation of the windowing logic. We promote the use of our runtime library due to its flexibility and its strict adherence to the operational semantics outlined in this paper. Furthermore, our library also includes a compile-time component to help with the code generation for the window configuration, to automate the step of configuring windows based on the SPL operator invocation. Despite all these, it may still be desirable, from the perspective of an operator developer, to use their own implementation. This is more applicable to operators that only support a very narrow range of window configurations that may benefit from highly specialized implementations such as an FFT operator that only supports count-based windows.

6. EVALUATION

We have implemented the windowing library outlined in this paper in both C++ and Java languages. Our experience implementing operators using it has been mostly on the C++ version of the library, which we report here. The C++ library is around 3800 lines of code (LOC), not counting comments and non-statements.

We have used the library to implement a relational operators toolkit, a time-series toolkit, and an approximate summarization toolkit. Here, we report our experience from the relational operators toolkit, as it stretches the boundaries of the windowing support in full.

We perform our evaluation in two parts. We first review the list of relational operators and how they make use of the windowing library. As part of this review, we also compare the LOC information for the implemented operators and compare them to the LOC information for the same set of operators that were implemented without the windowing library. The latter numbers are taken from InfoSphere Streams [5] v1.0 product code that uses the SPADE [15] language, where relational operators have the special status of being part of the language itself and implement the windowing logic on their own, without the use of a common library exposed to operator developers.

In the second part of our evaluation, we compare the runtime performance of the windowing library employed in SPL to the SPADE implementation that mixes operator and windowing library code.

6.1. Relational operators

We look at the aggregate, join, and sort operators. The selection and projection operations do not use windows.

6.1.1. Aggregate operator. The aggregate operator performs windowed aggregations. It has a single input port with a window punctuation mode of *WindowBound* and a single output port with a window punctuation mode of *Generating*. It implements all available window configurations, including different window types (tumbling versus sliding), different eviction and trigger policies (count, time, attribute-delta, and punctuation based), partitioned windows, and different partition

DOI: 10.1002/spe

eviction policies (partition count, tuple count, and age-based). In total, 52 possible variations are supported with respect to windowing configuration.

The aggregate operator registers for the window flush event for tumbling windows. For sliding windows, it registers for the window trigger event, before tuple eviction event, and initial full event. Furthermore, it relies on window summarizers for incremental computation and memory savings. The operator is implemented using 1988 LOC for SPADE versus only 924 LOC[‡] for SPL. The windowing library cuts the implementation effort by more than half (54%) in terms of the code size.

6.1.2. Join operator. The join operator matches tuples across two streams, within specified windows. It has two input ports, each having its own window configuration. Both input ports have a window punctuation mode of *Oblivious*. The join windows are always sliding and the trigger policy is always fixed as count-based with size 1. This is because the join operator implements streaming join semantics, where each new tuple arriving on a stream results in checking for matches within the window defined on the opposing stream. The join operator has a single output port with a window punctuation mode of *Generating*. It also supports optional output ports for implementing outer joins.

The join operator fully supports partitioned windows and multiplexed processing, as long as the types of the partition by attributes match for the two input ports, if present. A popular configuration is to have a one-sided partitioned join, where one of the windows has size 0, which is useful for implementing hash lookups. The join operator also supports outer joins, where tuples that are evicted from their windows without ever being matched during their stay are output as well (on a separate output port or on the single output port, depending on the choice). In total, 90 possible variations are supported with respect to window configuration.

The join operator registers for the after tuple insertion event (needed for indexing), window trigger event (needed for running the matching logic), and before tuple eviction event (needed for outer joins). The operator is implemented using 1167 LOC for SPADE versus only 667 LOC for SPL. Again, the windowing library cuts the implementation effort by almost half (43%) in terms of the code size.

6.1.3. Sort operator. The sort operator implements windowed sorting. The sort operator has a single input port with a window punctuation mode of *WindowBound* and a single output port with a window punctuation mode of *Generating*. It supports two modes of operation: *batch* and *progressive*. The batch mode of operation sorts one tumbling window at a time and supports all tumbling window configurations. The progressive mode performs a sort over a sliding window, where each newly arriving tuple on a full window results in outputting the next tuple in the window that is first in the sort order. This kind of progressive sort is useful for ordering streams that are unordered with a bound. In this mode, the sort operator supports only sliding windows with count-based trigger policies of size 1. In total, 44 possible variations are supported with respect to window configuration.

The sort operator registers for the before window flush event in the batch mode. For progressive sort, it registers for the after tuple insertion (to perform incremental sort) and before tuple eviction events (to produce the next result). The operator is implemented using 410 LOC for SPADE versus 284 LOC for SPL. In this case, the windowing library cuts the implementation effort by 30% in terms of the code size.

6.2. Runtime evaluation

We use an aggregate operator to compare the performance of SPL-based windowing library implementation to that of SPADE's hand-coded windowing logic that is intermixed with the operator implementation.

Figure 3 plots the throughput of SPL relative to SPADE as a function of window size for a tumbling window with different eviction policies for both partitioned and non-partitioned variants. The

[‡]The SPL version of the aggregate operator uses numerically sensitive methods for computing summaries, which takes an additional 500 lines of code and is excluded from the LOC count for SPL, as this capability is not present in SPADE.

DOI: 10.1002/spe

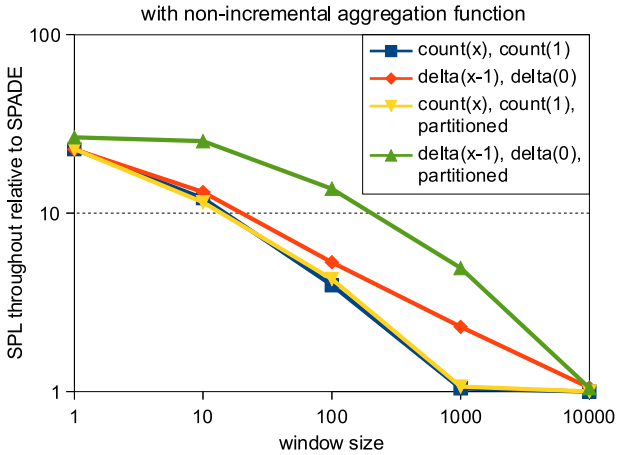

Figure 3. Throughput for aggregation using a tumbling window and an incremental aggregation (average).

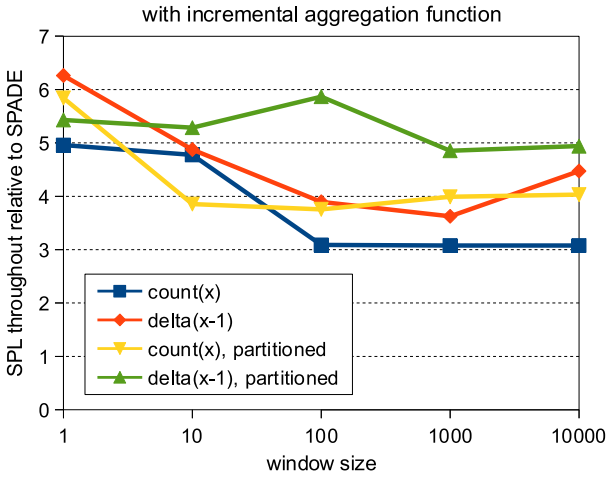

Figure 4. Throughput for aggregation using a sliding window and a non-incremental aggregation (percentile).

aggregation that was used for this experiment is `Average`, which is an incrementally computable one. The results are surprising, as the windowing library implementation performs around five times better in general. After investigating the code for the SPADE implementation, we have found that the operator implementers have opted for a design that avoids the code variation involved in supporting time-based and non-time-based policies by reducing all cases to the former. In this design, the window processing is always performed by a different thread than the one that performs tuple ingestion. On the other hand, our windowing library only uses this kind of separation for time-based policies and employs a streamlined solution that does not involve threads for the other cases.

Figure 4 plots the throughput of SPL relative to SPADE as a function of window size for a sliding window with different eviction policies for both partitioned and non-partitioned variants. The aggregation that was used for this experiment is `Percentile`, which is a non-incrementally computable one. We see that SPL's windowing library implementation shows significant improvement, but the gap narrows down eventually to zero as the window gets larger. We picked a non-incremental aggregation to show that no matter what the windowing library does, for aggregations that require processing the entire window the operator logic eventually dominates the overall cost.

The goal of these experiments is to show that separating the operator logic and windowing logic via the use of an event-based interface and policy-driven design does not introduce overhead. While a coupled implementation like SPADE's have the potential to be more efficient, in practice the

DOI: 10.1002/spe

complications that result from supporting a large number of window configurations in a coupled setup makes it difficult to optimize the code properly.

7. RELATED WORK

To our knowledge, there is no previous published work on the design and implementation of a general purpose windowing library for data stream processing systems. This can be attributed to the lack of extensibility in many of the existing systems with respect to creating new operators that are as capable as the so called 'built-in' ones. However, there is a large body of related work dealing with the semantics of windows as well as query evaluation techniques to implement select windowed operators.

It has been shown recently that not only is the windowing behavior significantly different across stream processing systems (exemplified by research [1–3, 18, 19] and commercial systems [4–6, 12]), but also the semantics are not always well-defined [20]. In particular, a descriptive model is introduced in [20], in order to analyze the behavior of stream processing systems, understand the results of windowed queries, and explain the key differences in windowing behavior across different systems.

To illustrate the subtle differences, we give two examples. As one example, some systems process partial windows whereas some systems only process full windows. Partial windows happen during start up. Our windowing library supports both options and leaves it to the operators to expose this as a parameter. As another example, different systems use differing techniques for assigning timestamps to tuples (system timestamps vs source timestamps). Our windowing library does not make any assumptions about this, and instead enables one to specify the attribute over which the delta based windows are defined. In summary, the subtleties of different systems can easily be supported on top of our windowing library.

The minor differences between the windowing semantics across various systems is so vast [20] that, it is beyond the scope of our work to characterize all such differences. It is important to note that none of these systems have decoupled the windowing functionality from the operator implementations. We make this decoupling in this work and provide windowing as a service to operators, so that user-defined operators with windowing support can be built.

Many of the stream processing systems that are based on SQL, exemplified by STREAM [1], Gigascope [18], and TelegraphCQ [19], provide various forms of windows to deal with the streaming versions of the traditionally blocking relational operators. Windows are represented via syntax extensions to SQL, and query semantics are extended to take into account windowed processing. Previous work has also looked at formally defining the semantics of streaming queries in the presence of windows [21, 22], but they do not cover windowing semantics independent of the operations performed as part of the query processing. The windowing library presented here covers all the window variations discussed in previous works (and more), while at the same time decoupling operator logic from the window management.

One exception is the work presented in [16], which clearly defines windowing semantics and then uses it to develop efficient techniques to evaluate aggregation queries. The various window configurations covered in [16] are also covered by our work. However, there are a few fundamental differences. First, our work provides operational semantics for windows, providing details on the windowing events and the order in which they are delivered. This is important for our work, as we aim at providing a general purpose library, where the main design point is the event-based interface used to decouple windowing logic from operator logic. The work presented in [16] focuses on domain-driven window semantics, providing a mapping between tuples and window ids. Second, our work defines windows in terms of eviction, trigger, and partition eviction policies. We are not aware of previous work that have looked at partition eviction policies. SQL-based systems often use the *range* and *slide* concepts in place of eviction and trigger policies and the coverage of semantics in the presence of mixed polices is limited in the literature. Finally, although punctuations [14] and their semantics [23] have been discussed in the literature, our definition of window punctuations, rules for their propagation in the presence of multi-input and multi-output operators, and their relationship to punctuation-based windows is unique. Our window punctuations can be seen as a special

DOI: 10.1002/spe

case of the more general definition used for punctuations in the literature (an ordered set of patterns [23]). Although they represent a structurally simple type of punctuation, our window punctuations have unique semantics that are associated with window processing and have been used successfully in a commercial stream processing language [11].

Various existing works address efficient implementation of windowed operators, such as joins [8, 24] and aggregations [7, 25]. These are complimentary to our work, which focuses on a general purpose windowing library for stream processing.

8. CONCLUSION

We presented the design and implementation of a windowing library for extensible stream processing systems. We described windowing types and policies commonly used in stream processing applications and provided operational semantics for all window configurations. Our windowing library facilitates the implementation of user-defined operators with windowing logic by decoupling the operator logic from the details of window handling, through an event-driven interface and policy-based design. We showed the efficacy of the library based on our experience of using it in a commercial grade stream processing middleware to implement stream relational operators.

REFERENCES

1. Arasu A, Babcock B, Babu S, Datar M, Ito K, Motwani R, Nishizawa I, Srivastava U, Thomas D, Varma R, Widom J. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin* 2003; **26**(1):19–26.
2. Abadi D, Ahmad Y, Balazinska M, Çetintemel U, Cherniack M, Hwang J-H, Lindner W, Maskey A, Rasin A, Ryvkina E, Tatbul N, Xing Y, Zdonik S. The design of the Borealis stream processing engine. In *Innovative Data Systems Research Conference (CIDR)*, Asilomar, CA, USA, 2005; 277–289.
3. Jain N, Amini L, Andrade H, King R, Park Y, Selo P, Venkatramani C. Design, implementation, and evaluation of the linear road benchmark on the Stream Processing Core. In *International Conference on Management of Data (ACM SIGMOD)*, Chicago, IL, USA, 2006; 431–442.
4. StreamBase Systems. (Available from: http://www.streambase.com) [retrieved October, 2011].
5. Gedik B, Andrade H. A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere Streams. *Software: Practice and Experience* 2012; **42**(11):1363–1391.
6. S4 stream computing platform. (Available from: http://www.s4.io/) [retrieved October, 2011].
7. Golab L, Bijay KG, Özsu MT. Multi-query optimization of sliding window aggregates by schedule synchronization. In *Conference on Information and Knowledge Management (ACM CIKM)*, Arlington, Virginia, USA, 2006; 844–845.
8. Gedik B, Wu K-L, Yu PS, Liu L. GrubJoin: an adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *Transactions on Knowledge and Data Engineering (IEEE TKDE)* 2007; **19**(10):1363–1380.
9. Wu K-L, Yu PS, Gedik B, Hildrum K, Aggarwal CC, Bouillet E, Fan W, George D, Gu X, Luo G, Wang H. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Very Large Data Bases Conference (VLDB)*, Vienna, Austria, 2007; 1185–1196.
10. Turaga D, Andrade H, Gedik B, Venkatramani C, Verscheure O, Harris D, Cox J, Szewczyk W, Jones P. Design principles for developing stream processing applications. *Software: Practice and Experience* 2010; **40**(12):1073–1104.
11. Hirzel M, Andrade H, Gedik B, Kumar V, Losa G, Mendell M, Nasgaard H, Soulé R, Wu K-L. SPL language specification. *Tecnical Report RC24897*, IBM, Yorktown Heights, New York, USA, 2009.
12. Ali MH, Chandramouli B, Raman BS, Katibah E. Spatio-temporal stream processing Microsoft StreamInsight. *IEEE Data Engineering Bulletin* 2010; **33**(2):69–74.
13. Andrade H, Gedik B, Wu K-L, Yu PS. Processing high data rate streams in System S. *Journal of Parallel and Distributed Computing* 2011; **71**(2):145–156.
14. Maier D, Tucker PA. Punctuations. In *Encyclopedia of Database Systems*. Springer, 2009; 2216–2217.
15. Gedik B, Andrade H, Wu K-L, Yu PS, Doo M. Spade: The System S declarative stream processing engine. In *International Conference on Management of Data (ACM SIGMOD)*, Vancouver, Canada, 2008; 1123–1134.
16. Li J, Maier D, Tufte K, Papadimos V, Tucker PA. Semantics and evaluation techniques for window aggregates in data. In *International Conference on Management of Data (ACM SIGMOD)*, Baltimore, Maryland, USA, 2005; 311–322.
17. Aggarwal CC. A survey of synopsis construction in data streams. In *Data Streams*. Springer, 2007; 169–207.
18. Cranor CD, Johnson T, Spatscheck O, Shkapenyuk V. Gigascope: a stream database for network applications. In *International Conference on Management of Data (ACM SIGMOD)*, San Diego, CA, USA, 2003; 647–651.
19. Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, Shah MA. TelegraphCQ: continuous dataflow processing for an uncertain world. In *Innovative Data Systems Research Conference (CIDR)*, Asilomar, CA, USA, 2003; 269–280.

DOI: 10.1002/spe

20. Botan I, Derakhshan R, Dindar N, Haas L, Miller RJ, Tatbul N. SECRET: a model for analysis of the execution semantics of stream processing systems. In *Very Large Data Bases Conference (VLDB)*, Singapore, 2010; 232–243.
21. Arasu A, Babu S, Widom J. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Databases (VLDBJ)* 2006; **15**(2):121–142.
22. Kramer J, Seeger B. Semantics and implementation of continuous sliding window queries over data streams. *Transactions on Database Sytems (ACM TODS)* 2009; **34**(1):1–49.
23. Tucker PA, Maier D, Sheard T, Fegaras L. Exploiting punctuation semantics in continuous data streams. *Transactions on Knowledge and Data Engineering (IEEE TKDE)* 2003; **15**(3):555–568.
24. Srivastava U, Widom J. Memory-limited execution of windowed stream joins. In *Very Large Data Bases Conference (VLDB)*, Toronto, Canada, 2004; 324–335.
25. Arasu A, Widom J. Resource sharing in continuous sliding-window aggregates. In *Very Large Data Bases Conference (VLDB)*, Toronto, Canada, 2004; 336–347.

DOI: 10.1002/spe