SAS® Event Stream Processing 6.2: Overview

What Is SAS Event Stream Processing?

Product Overview

SAS Event Stream Processing enables you to quickly process and analyze a large number of continuously flowing events. Events are delivered through high throughput, low latency data flows called *event streams*. Those event streams are published in applications that use the following:

- connector classes or adapter executables
- the C, JAVA, or Python publish/subscribe APIs
- SAS Event Stream Processing Streamviewer
- SAS Event Stream Processing Studio

You can embed event stream processing engines with dedicated thread pools within new or existing applications that you write in XML. You can use the ESP client to feed event stream processing engine definitions (called projects) into an ESP server. In Jupyter Lab, you can use ESPPy, an open-source package for Python, to program event stream processing models and connect to an ESP server.

Event stream processing applications can perform real-time analytics on streams of events. Typical use cases for event stream processing include but are not limited to the following:

- sensor data monitoring and management
- operational systems condition monitoring and management
- object detection and classification
- cyber security analytics
- capital markets trading systems
- fraud detection and prevention
- personalized marketing

There are three variants of SAS Event Stream Processing.

Table 1 *Product Variants*

Product	Description
SAS Event Stream Processing	This is the core product. In addition to the ESP server and ESP client , it provides the following components: - SAS Event Stream Processing Analytics, which enables you to use advanced analytics and machine learning techniques in an event stream processing project. For more information, see *SAS Event Stream Processing: Using Streaming Analytics*

Product	Description
	■ SAS Event Stream Processing Studio, a web-based client that enables you to create, edit, upload, and test event stream processing projects. For more information, see *SAS Event Stream Processing: Using SAS Event Stream Processing Studio* ■ SAS Event Stream Processing Streamviewer, a web-based dashboard tool that you can use to visualize events as they flow through a project. For more information, see *SAS Event Stream Processing: Using SAS Event Stream Processing Streamviewer*
SAS Event Stream Processing for Edge Computing	This product provides a configurable disk footprint for simplified deployment of components to smaller edge systems. Use SAS Mirror Manager, a command-line utility for synchronizing a collection of SAS software repositories, to install it. Using this utility, you select packages to install in order to control the size of the deployment on the edge device. You must install the contents of the **`/basic`** directory. The following directories provide optional additions to functionality: ■ **`/analytics`**: contains analytical algorithms that are packaged with SAS Event Stream Processing. For more information, see *SAS Event Stream Processing: Using Streaming Analytics*. ■ **`/astore`**: contains libraries and extensions that offline models contained in analytic store files require for execution. For more information about those models, see "Loading Models Stored in Analytic Store Files " in *SAS Event Stream Processing: Using Streaming Analytics*. ■ **`/gpu`**: contains software that enables you to use a GPU to run an offline model contained in analytic store file. For information about how to do this, see "Processing Loaded Analytic Store Files with GPUs" in *SAS Event Stream Processing: Using Streaming Analytics*. ■ **`/textanalytics`**: contains software to enable the use of the Text Topic window and speech to text functionality. For more information, see "Using Text Topic Windows" in *SAS Event Stream Processing: Using Source and Derived Windows*. For more information about SAS Event Stream Processing for Edge Computing, see *SAS Event Stream Processing for Edge Computing: Deployment Guide*.
SAS Event Stream Processing for CAS	This version works alongside SAS Cloud Analytic Services (CAS) and provides the ability to query event stream projects and to run models in a cluster. It uses the loadStreams action set, which is automatically installed on all systems with SAS Viya: This action set provides actions to query event stream processing projects and get snapshots of streaming data. Single–pass actions can directly consume event stream processing data and perform analyses without first putting the streaming data into a table

Product	Description
	For more information, see "Using SAS Event Stream Processing with SAS Cloud Analytic Services Actions" in *SAS Event Stream Processing: Using SAS Event Stream Processing with Other Applications*.

SAS Event Stream Processing is shipped with SAS Event Stream Manager, a web-based client that enables you to manage your SAS Event Stream Processing environment. For more information, see *SAS Event Stream Manager: Using SAS Event Stream Manager*.

You can deploy SAS Event Stream Processing in a test environment with a "rapid deployment," or deploy it with SAS Viya. You can also obtain pre-built Docker images to deploy in a Kubernetes cluster. For more information, see *SAS Event Stream Processing on Linux: Deployment Guide*

Planning Your Event Stream Processing Application

Conceptually, an event is something that happens at a determinable time that can be recorded as a collection of fields. As you plan your event stream processing application, answer the following questions:

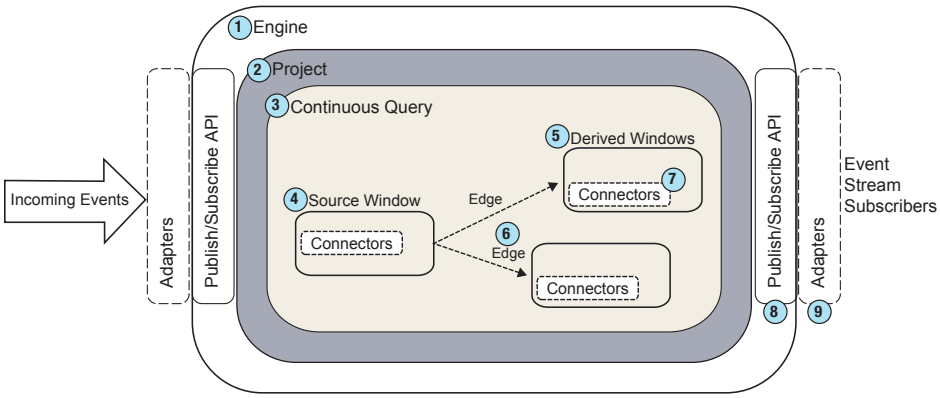
- What specific event streams are published into an application, and with what protocol and format?
- What happens to the data? That is, how are event streams transformed and analyzed?
- What are the resulting event streams of interest? What applications subscribe to these event streams, and in what format and protocol?

Your answers to these questions enable you to plan the structure of your event stream processing model.

What is an Event Stream Processing Model?

An event stream processing *model* specifies how input event streams from publishers are transformed and analyzed into meaningful resulting event streams consumed by subscribers. The following figure depicts the model hierarchy.

Figure 1 *The Event Stream Processing Model Hierarchy*



1 At the top of the model hierarchy is the *engine*. Each model contains only one engine instance with a unique name. The ESP server is an engine instance.

2 The engine contains one or more *projects*, each uniquely named. Projects run in a dedicated *thread pool* whose size is defined as a project attribute. You can specify a port so that projects can be spread across network interfaces for throughput scalability. Using a pool of threads in a project enables the event stream processing engine to use multiple processor cores for more efficient parallel processing.

3 A project contains one or more *continuous queries*. A continuous query is represented by a directed graph. This graph is a set of connected nodes that follow a direction down one or more parallel paths. Continuous queries are data flows, which are data transformations and analysis of incoming event streams.

4 Each query has a unique name and begins with one or more *source windows*.

5 Source windows are typically connected to one or more *derived windows*. Derived windows can detect patterns in the data, transform the data, aggregate the data, analyze the data, or perform computations based on the data. They can be connected to other derived windows.

6 Windows are connected by *edges*, which have an associated direction. In this context, edges are a program element that specifies connectivity between two or more windows.

7 *Connectors* publish or subscribe event streams to and from an engine. Connectors are in-process to the engine.

8 The *publish/subscribe API* can be used to subscribe to an event stream window either from the same machine or from another machine on the network. Similarly, the publish/subscribe API can be used to publish event streams into a running event stream processor project Source window.

9 *Adapters* are stand-alone executable programs that can be networked. Adapters use the publish/subscribe API to publish event streams to do the following:

- publish event streams to Source windows
- subscribe to event streams from any window

Several objects in the modeling layers measure time intervals in microseconds. The following intervals are measured in milliseconds:

- time-out period for patterns
- retention period in time-based retention
- pulse intervals for periodic window output

Most non-real-time operating systems have an interrupt granularity of approximately 10 milliseconds. Thus, specifying time intervals smaller than 10 milliseconds can lead to unpredictable results.

Note: In practice, the smallest value for these intervals should be 100 milliseconds. Larger values give more predictable results.

Understanding Events

What is an Event?

An event is an individual record of an event stream. It is the fundamental building block of event stream processing. An event consists of metadata and field data.

An event's metadata consists of the following:

- an operation code (opcode)
- a set of flags (indicating whether the event is a normal, partial-update, or a retention-generated event from retention policy management)
- a set of four microsecond timestamps that can be used for latency measurements

Table 2 *Opcodes Supported by SAS Event Stream Processing*

Opcode	Description
Delete (D)	Removes event data from a window
Insert (I)	Adds event data to a window
Update (U)	Changes event data in a window
Upsert (P)	Updates event data if the key field already exists. Otherwise, it adds event data to a window.
Safe Delete (SD)	Removes event data from a window without generating an error if the event does not exist.

One or more fields of an event must be designated as a primary key. Key fields enable the support of opcodes.

Data in an event object is stored in an internal format as described in the schema object . All key values are contiguous and packed at the front of the event. An event object maintains internal hash

values based on the key with which it was built. In addition, there are functions in the `dfESPeventcomp namespace` for a quick comparison of events that were created using the same underlying schema.

When publishing, when you do not know whether an event needs an Update or Insert opcode, use Upsert. The Source window where the event is injected determines whether it is handled as an Insert or an Update. The Source window then propagates the correct event and opcode to the next set of connected windows in the model or to subscribers.

Data Types in Events

Events support the following data types:

- INT32
- INT64
- DOUBLE
- STRING
- DATE (granularity to the second)
- STAMP (granularity to the microsecond)
- MONEY (192–bit fixed decimal)
- BINARY (binary large object or blob)
- RUTF8STR (reference-counted string or rstring)
- ARRAY (32–bit integers, 64–bit integers, double)

With the base data types (INT32, INT64, DOUBLE, STRING, DATE, STAMP, and MONEY), data is stored inline in the event. Inline storage enables fast indexing and serialization.

The BINARY, RUTF8STR, and ARRAY data types are not stored inline. Instead, they are referenced in an event, which means that the event holds a pointer to the actual data. These data types cannot be used as key fields for an event. They are reference-counted at the object level. This enables the same object to be referenced in multiple events, which reduces memory usage and the time it takes to create a new object and copy the data.

Note: Missing or null values are not supported in arrays. When the ESP server parses a CSV file that contains an array with an omitted value (for example, `[1.0;;3.0]`), it fills the omission with a 0 of the appropriate data type.

Note: NaN (not a number) is permitted in arrays of the double data type.

For non-key fields, the RUTF8STR data type might be more economical than STRING. RUTF8STR can be passed in and out of all windows. It is internally referenced as a standard string whenever you use it. These characteristics can lead to considerable savings in memory. Remember that a STRING data type is stored inline in an event. When a 16K string is propagated through a chain of windows, a copy of the string is included in the events in each of the windows. When you use RUTF8STR

instead, the first use creates a dfESPrstring object that holds the string. Each event contains an 8–byte pointer to that object.

Converting CSV Events into Binary Code

You can convert a file or stream of CSV events into a file or stream of binary events. This file or stream can be published into a project and processed at higher rates than the CSV file or stream.

CSV conversion to binary is very CPU intensive, so it is recommended to convert files one time or convert streams at the source. In actual production applications, the data frequently arrives in some type of binary form and needs only reshuffling to be used in SAS Event Stream Processing. Otherwise, the data comes as text that must be converted to binary.

To properly represent string fields in an event, the corresponding CSV string field must follow these rules:

- When a string field includes leading or trailing white space, you must enclose the entire string field in double quotation marks
- When a string field includes the CSV delimiter character (which is ',' by default), you must enclose the entire string field in double quotation marks.
- You must prefix literal double quotation mark (") characters in a string field with a leading escape character ('\').
- You must prefix literal escape ('\') characters in a string field with a leading escape character ('\').
- The multi-byte byte-order mark (BOM) sequence is unsupported. If you include it, it prevents proper parsing of a CSV string.
- The new line ('\n') sequence is not supported. CSV is a line-oriented format and new line is reserved as the line delimiter.

Understanding Event Blocks

Event blocks contain zero or more binary events. Publish/subscribe clients send and receive event blocks to or from the engine. Because publish/subscribe operations carry overhead, working with event blocks that contain multiple events (for example, 512 events per event block) improves throughput performance with minimal impact on latency.

Table 3 *Event Block Types*

Event Block	Description
Transactional	Processing through the project is atomic. If one event in the event block fails (for example, deleting a non-existing event), then all of the events in the event block

Event Block	Description
	fail. Events that fail are logged and placed in an optional bad records file, which can be processed further.
Normal	Processing through the project is not atomic. Events are packaged together for efficiency, but are individually treated once they are injected into a source window.

A unique transaction ID is propagated through transformed event blocks as the blocks work their way through an engine model. This persistence enables event stream subscribers to correlate a group of subscribed events back to a specific group of published events through the ID.

Typically, you set the size of an event block with the `blocksize` parameter of the connector or adapter used in your project. To optimize engine performance, the following is recommended.

- For small events (>=1 and <= 6 fields, contains no blobs), set `blocksize` to 256.
- For medium events (>=7 and <=12 fields, contains no blobs), set it to 32.
- For large events(>=13 fields, contains no blobs), set it to 4.
- For huge events (contains blobs), set it to 1.

There is rarely justification to set `blocksize` larger than 16000. That value is practical only when small events arrive very quickly. Whenever events include blob data for images, set `blocksize` to 1.

Implementing Engines

An engine is the top level container in the event stream processing model hierarchy. Each model contains only one engine instance with a unique name. Engines can be instantiated as stand-alone executables or embedded within an application.

SAS Event Stream Processing provides two ways to implement engines:

- The ESP server enables you to define single engine definitions and to define an engine with dynamic project creations and deletions. You can use the ESP server with other products such as SAS Visual Investigator. For more information, see *SAS Event Stream Processing: Using the ESP Server*.
- The C++ Modeling API enables you to embed an event stream processing engine inside an application process space. It also provides low-level functions that enable an application's main thread to interact directly with the engine. For more information, open **`$DFESP_HOME/doc/html/index.html`** (UNIX deployments) or **`%DFESP_HOME%\doc\html\index.html`** (Windows deployments) in a web browser. This provides access to the complete class and method documentation.

Deciding whether to implement multiple projects or multiple continuous queries depends on your processing needs. For the ESP server, multiple projects can be dynamically introduced, destroyed, stopped, or started because the layer is being used as a service. For all modeling layers, multiple

projects can be used for different use cases or to obtain different threading models in a single engine instance. You can use:

- a single-threaded model for a higher level of determinism
- a multi-threaded model for a higher level of parallelism

Because you can use continuous queries as a mechanism of modularity, the number of queries that you implement depends on how compartmentalized your windows are. Within a continuous query, you can instantiate and define as many windows as you need. Any given window can flow data to one or more windows. Loop-back conditions are not permitted within continuous queries. You can loop back across continuous queries using the project connector or adapter.

Event streams must be published or injected into Source windows through one of the following:

- the publish/subscribe API
- connectors
- adapters
- HTTP clients
- SAS Event Stream Processing Studio
- SAS Event Stream Processing Streamviewer
- the continuous-query-inject method in the C++ Modeling API

Within a continuous query, you can define a data flow model using all of the available window types. Procedural windows enable you to write event stream input handlers using C++.

Understanding Projects

A *project* specifies a container that holds one or more continuous queries and is backed by a thread pool of user-defined size. The level of determinism for a project's incremental computations is, by default, full concurrency. You can change this with the `use-tagged-token` attribute of the `project` element, which enables a project to use tagged token data flow semantics. You can also specify an optional port for publish/subscribe scalability.

The data flow model is always computationally deterministic. When a project is multi-threaded, intermediate calculations can occur at different times across different project runs. Therefore, when a project watches every incremental computation, the increments could vary across runs even though the unification of the incremental computation is always the same.

Note: Regardless of the determinism level or of the number of threads used in the engine, each window always processes all data in order. Therefore, data received by a window is never rearranged and processed out of order.

Understanding Continuous Queries

A continuous query specifies a container that holds one or more directed graphs of windows and that enables you to specify the connectivity between windows. The windows within a continuous query can transform or analyze data, detect patterns, or perform computations. Query containers provide functional modularity for large projects. Typically, each container holds a single directed graph.

Continuous query processing follows these steps:

1. An event block (with or without atomic properties) that contains one or more events is injected into a Source window.
2. The event block flows to any derived window that is directly connected to the Source window. If transactional properties are set, then the event block of one or more events is handled atomically as it makes its way to each connected derived window. That is, all events must be performed in their entirety.

   If any event in the event block with transactional properties fails, then all of the events in the event block fail. Failed events are logged. They are written to a bad records file for you to review, fix, and republish when you enable this feature.
3. Derived windows transform events into zero or more new events that are based on the properties of each derived window. After new events are computed by derived windows, they flow farther down the model to the next level of connected derived windows, where new events are potentially computed.
4. This process ends for each active path down the model for a given event block when either of the following occurs:
   - There are no more connected derived windows to which generated events can be passed.
   - A derived window along the path has produced zero resulting events for that event block. Therefore, it has nothing to pass to the next set of connected derived windows.

Understanding Windows

A continuous query contains one or more *Source windows* and one or more *derived windows*. All event streams must enter continuous queries by being published or injected into a Source window. Event streams cannot be published or injected into any other window type.

Windows are connected by *edges*, which have an associated direction.

SAS Event Stream Processing supports the following derived window types:

Table 4 *Derived Window Types*

Window Type	Description
Compute window	Enables a one-to-one transformation of input events into output events through the computational manipulation of the input event stream fields.
Aggregate window	Similar to a Compute window in that non-key fields are computed. An Aggregate window uses the key field or fields for the group-by condition. All unique key field combinations form their own group within the Aggregate window. All events with the same key combination are part of the same group.
Copy window	Makes a copy of the parent window. Making a copy can be useful to set new event state retention policies. Retention policies can be set only in source and Copy windows. You can set event state retention for a Copy window only when the window is not specified to be Insert-only and when the window index is not set to `pi_EMPTY`. All subsequent sibling windows are affected by retention management. Events are deleted when they exceed the windows retention policy.
Counter window	Enables you to see how many events are streaming through your model and the rate at which they are being processed.
Filter window	Uses a registered Boolean filter function or expression. This function or expression determines what input events are allowed into the Filter window.
Functional window	Enables you to use different types of functions to manipulate or transform the data in events. Fields in a Functional window can be hierarchical, which can be useful for applications such as web analytics.
Geofence window	Enables you to create a window to determine whether the location of an event stream is inside or near an area of interest.
Join window	Takes two input windows and a join type. Supports equijoins that are one to many, many to one, or many to many. Both inner and outer joins are supported.
Notification window	Enables you to send notifications through email, text, or multimedia message. You can create any number of delivery channels to send the notifications. A Notification window uses the same underlying language and functions as the Functional window.
Object Tracking window	Enables you to perform multi-object tracking (MOT) in real time.
Pattern window	Enables the detection of events of interest (EOI). A pattern defined in this window type is an expression that logically connects declared events of interest. To define a pattern window, you need to define events of interests and then connect these events of interest using operators. The supported operators are

Window Type	Description
	"AND", "OR", "FBY", "NOT", "NOTOCCUR", and "IS". The operators can accept optional temporal conditions.
Procedural window	Enables the specification of an arbitrary number of input windows and input-handler functions for each input window (that is, event stream).
Remove State window	Facilitates the transition of a stateful part of a model to a stateless part of a model.
Text Category window	Enables you to categorize a text field in incoming events. A Text Category window is Insert-only. The text field could generate zero or more categories with scores. This object enables users who have licensed SAS Contextual Analysis to use its MCO files to initialize a Text Category window.
Text Context window	Enables the abstraction of classified terms from an unstructured string field. This object enables users who have licensed SAS Contextual Analysis to use its Liti files to initialize a Text Context window. Use this window type to analyze a string field from an event's input to find classified terms. Events generated from those terms can be analyzed by other window types. For example, a Pattern window could follow a text context window to look for tweet patterns of interest.
Text Sentiment window	Determines the sentiment of text in the specified incoming text field and the probability of its occurrence. The sentiment value is "positive," "neutral," or "negative." The probability is a value between 0 and 1. A Text Sentiment window is Insert-only. This object enables users who have licensed SAS Sentiment Analysis to use its SAM files to initialize a Text Sentiment window.
Text Topic window	Run SAS Text Miner analytics on events. Text topic windows receive and process text from documents as string fields. Text mining analytics models enter a text topic window through an analytic store file.
Transpose window	Enables you to interchange an event's rows as columns, or columns as rows.
Union window	Combines multiple event streams with the same schema into a single stream, similar to an SQL union operation.

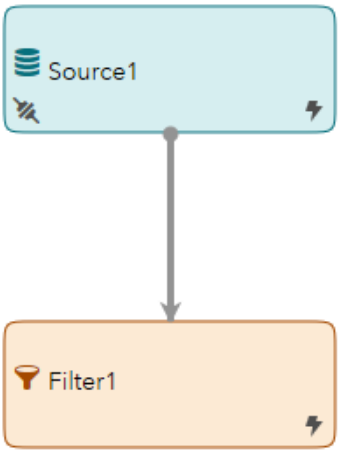
SAS Event Stream Processing Analytics provides an additional set of window types. For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

Streaming Events through a Continuous Query

Overview

The following example demonstrates what happens when events stream through the windows in a continuous query:

Figure 2 *Continuous Query with a Source Window and a Filter Window*



In this query, a Source window publishes events to a single Filter window. The Source window contains the following schema:

```
ID*: int32, symbol: string, quantity: int32, price: double
```

This schema consists of four fields:

Table 5 *Source Window Schema*

Field	Type
ID	32-bit integer
symbol	string
quantity	32-bit integer

Field	Type
price	double precision floating-point

Key fields in a schema identify an event for operations such as Insert, Update, Delete, or Upsert. Key fields must be unique. If you think of the event stream as a database, then you can think of the key fields as lookup keys.

In the schema, the ID field has the * designator to indicate that this field is part of the key. No other field in the schema has this designator, so the ID field completely forms the key.

Here is XML code for the Source window:

```
<window-source name="Source1" pubsub="true">
      <schema>
        <fields>
          <field type="int32" name="ID" key="true" />
          <field type="string" name="symbol" />
          <field type="int32" name="quantity" />
          <field type="double" name="price" />
        </fields>
      </schema>
      <connectors>
        <connector class="fs" name="New_Connector_1">
          <properties>
            <property name="type">pub</property>
            <property name="fsname">events.csv</property>
            <property name="fstype">csv</property>
            <property name="transactional">true</property>
            <property name="blocksize">1</property>
          </properties>
        </connector>
      </connectors>
</window-source>
```

A file and socket connector publishes a CSV file in the current directory that contains the events to be processed into the Source window.

The Filter window applies the expression `quantity > 1000`. Thus, events are passed only when the **`Quantity`** field in the event exceeds the value of 1000.

Here is XML code for the Filter window:

```
<window-filter name="Filter1" pubsub="true">
      <expression>quantity > 1000</expression>
</window-filter>
```

Here is the XML for the edge that connects the Source window to the Filter window:

```
<edges>
    <edge target="Filter1" source="Source1" />
 </edges>
```

For more information about available filter conditions, see "Overview to Expressions" in *SAS Event Stream Processing: Using Source and Derived Windows*. For information about the XML language that you use to code continuous queries, see *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

The following sections provide detailed information about what happens when five events stream through this model.

Processing the First Event

Suppose that the first event streaming through the query is as follows:

```
e1: [i,n,10,IBM,2000,164.1]
```

1 The Source window receives `e1` as an Input event. It stores the event and passes it to the Filter window.

2 The Filter window receives `e1` as an Input event, as designated by the "i" in the first field. The second field in this and all subsequent events designates "normal."

3 The **`Quantity`** field has a value of 2000. Because the filter expression is `quantity > 1000`, the Filter window stores the input. Typically, a Filter window would pass `e1` forward. However, because the Filter window has no dependent windows, there is no additional data flow for the event.

The window contents are now as follows:

Table 6 *Contents of Source Window after First Event*

ID	Symbol	Qty	Price
10	IBM	2000	164.10

Table 7 *Contents of Filter Window after First Event*

ID	Symbol	Qty	Price
10	IBM	2000	164.10

Processing the Second Event

The second event is as follows:

```
e2: [p,n,20,MSFT,1000,114.22]
```

1 The Source window receives `e2` as an Upsert event. It checks whether the window has a stored event with a key (ID) of 20.

2 An ID of 20 is not stored, so the Source window creates a new event `e2a: [I, 20, "MSFT", 1000, 114.22]`. It stores this new event and passes it to the Filter window.

3 The Filter window receives `e2a` as an Input event.

4 The value in the **`Quantity`** field of `e2` equals 1000, which does not meet the condition set by the filter expression in the schema. Thus, this event is not stored or passed to any dependent windows.

The window contents are now as follows:

Table 8 *Contents of Source Window after Second Event*

ID	Symbol	Qty	Price
10	IBM	2000	164.10
20	MSFT	1000	114.22

Table 9 *Contents of Filter Window after Second Event*

ID	Symbol	Qty	Price
10	IBM	2000	164.10

Processing the Third Event

The third event is as follows:

```
e3: [d,n,10, , , , ]
```

Note: For a Delete event, you need only specify key fields. Remember that in this example, only the ID field is key.

1 The Source window receives `e3` as a Delete event.

2 The Source window looks up the event that is stored with the same key. The Delete opcode removes the event from the Source window.

3 The Source window passes the found record to the Filter window with the Delete opcode specified. In this case, the record that is passed to the Filter window is as follows:

```
e3a: [d,n,10,IBM,2000,164.1]
```

4 The Filter window receives `e3a` as an Input event.

5 The value in the **`Quantity`** field of `e3a` equals 2000. This old event that was previously stored makes it through the filter, so it is removed.

The window contents are now as follows:

Table 10 *Contents of Source Window after Third Event*

ID	Symbol	Qty	Price
20	MSFT	1000	114.22

The Filter window is empty.

Processing the Fourth Event

The fourth event is as follows:

```
e4: [u,n,20,MSFT,3000,114.25]
```

1. The Source window receives `e4` as an Update event.
2. The Source window looks up the event stored with the same key and modifies it.
3. The Source window constructs an update block that consists of the following:
   - the new record with updated values marked as an update block
   - the old record that was updated
4. The block is marked as a Delete event. The new event Update block that is passed to the Filter window looks like this:

   ```
   e4a: [ub,n,20,MSFT,3000,114.22] , [d,n,20,MSFT,1000,114.25]
   ```

   Note: Both the old and new records are supplied because derived windows often require the current and previous state of an event. They need these states in order to compute any incremental change caused by an Update.

5. The Filter window receives `e4a` as an Input event.
6. The value in the **`Quantity`** field of `e4a` > 1000, but previously it was <= 1000. The input did not pass the previous filter condition, but now it does pass. Because the input is not present in the filter window, the Filter window generates an Insert event of the following form:

   ```
   e4b: [i,n,20,MSFT,3000,114.25]
   ```

7. The Insert event is stored. The Filter window would pass `e4b`. However, because there are no dependent windows, this input does not pass. There is no further data flow for this event.

The window contents are now as follows:

Table 11 *Contents of Source Window after Fourth Event*

ID	Symbol	Qty	Price
20	MSFT	3000	114.25

Table 12 *Contents of Filter Window after Fourth Event*

ID	Symbol	Qty	Price
20	MSFT	3000	114.25

Processing the Fifth Event

The fifth event is as follows:

```
e5: [i,n,30,APPL,2000,225.06]
```

1 The Source window receives `e5` as an Insert event, stores it, and passes `e1` to the Filter window.

2 The Filter window receives `e5` as an Input event. Because the value in the **`Quantity`** field > 1000, the Filter window stores the input. Because the filter window has no dependent windows, there is no further data flow.

The window contents are now as follows:

Table 13 *Contents of Source Window after Fifth Event*

ID	Symbol	Qty	Price
20	MSFT	3000	114.25
30	APPL	2000	225.06

Table 14 *Contents of Filter Window after Fifth Event*

ID	Symbol	Qty	Price
20	MSFT	3000	114.25
30	APPL	2000	225.06

Code for the Example

The complete XML code that implements this example is available online in the **`xml`** directory as filter_exp_xml. The example provides the CSV file that contains data to stream these five events.

Developing an Event Stream Processing Application

1 Design, test, and validate an event stream processing model using SAS Event Stream Processing Studio. For more information, see *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

2 Run the model in the ESP server or within a stand-alone event stream processing application.

For more information about the ESP server, see "Setting Up and Using the ESP Server" in *SAS Event Stream Processing: Using the ESP Server*.

For more information about the C++ modeling objects to use to write a stand-alone event stream processing application, open **`$DFESP_HOME/doc/html/index.html`** (UNIX deployments) or **`%DFESP_HOME%\doc\html\index.html`** (Windows deployments) in a web browser. This provides access to the complete class and method documentation.

3 Publish one or more event streams into the engine one of the following ways:

- through connectors (in-process classes) or adapters (networked executables)

  Connectors are C++ classes that are instantiated in the same process space as the event stream processor. You can use connectors within XML models or C++ models. Adapters use the corresponding connector class to provide stand-alone executables that use the publish/subscribe API. Therefore, they can be networked. For more information, see "What Are Connectors and Adapters?" in *SAS Event Stream Processing: Connectors and Adapters*.

- through the Java, C, or Python publish/subscribe API

  For more information about the publish/subscribe API, see "Overview to the Publish/Subscribe APIs" in *SAS Event Stream Processing: Publish/Subscribe API Reference*.

- using the `dfESPcontquery::injectEventBlock()` method for C++ models

4 Subscribe to relevant window event streams within continuous queries using connectors, adapters, the publish/subscribe API, SAS Event Stream Processing Studio, SAS Event Stream Processing Streamviewer, or by using the `dfESPwindow::addSubscriberCallback()` method for C++ models.

For more information about SAS Event Stream Processing Streamviewer, see *SAS Event Stream Processing: Using SAS Event Stream Processing Streamviewer*.

You can use SAS Event Stream Manager, a web-based client, to do the following:

- deploy projects into test environments or production environments or both
- view the component parts of each deployment
- monitor the health of your deployments
- administer your deployments and manage change
- monitor your SAS Event Stream Processing metering servers

For more information, see *SAS Event Stream Manager: Using SAS Event Stream Manager*.

When you start a stand-alone C++ event stream processing application with **-b *filename***, the application writes the events that are not processed because of computational failures to the named log file. When you do not specify this option, the same data is written to `stderr`. It is recommended to create logs of bad events so that you can monitor them for new insertions.

Examples

Step by step examples for SAS Event Stream Processing and SAS Event Stream Manager are available at the SAS Event Stream Processing product support page.

SAS Event Stream Processing code examples are available online. Examples are written in the following programming languages.

Table 15 *Examples By Programming Language*

Programming Language	Subdirectory
XML	**xml**
Python	**python**
Java	**java**