SAS® Event Stream Processing 4.2: Overview

Product Overview

SAS Event Stream Processing enables programmers to build applications that can quickly process and analyze a large number of continuously flowing events. Programmers can build applications using the XML Layer or the C++ Modeling API that are included with the product. Event streams are published in applications using the C, JAVA, or Python publish/subscribe APIs, connector classes, adapter executables, Streamviewer, or SAS Event Stream Processing Studio.

Event stream processing engines with dedicated thread pools can be embedded within new or existing applications. The XML client can be used to feed event stream processing engine definitions (called projects) into an event stream processing XML server.

Event stream processing applications typically perform real-time analytics on streams of events. These streams are continuously published into an event stream processing engine. Typical use cases for event stream processing include but are not limited to the following:

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

Event stream processing enables the user to analyze continuously flowing data over long periods of time where low-latency incremental results are important.

Designing an Event Stream Processing Application

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

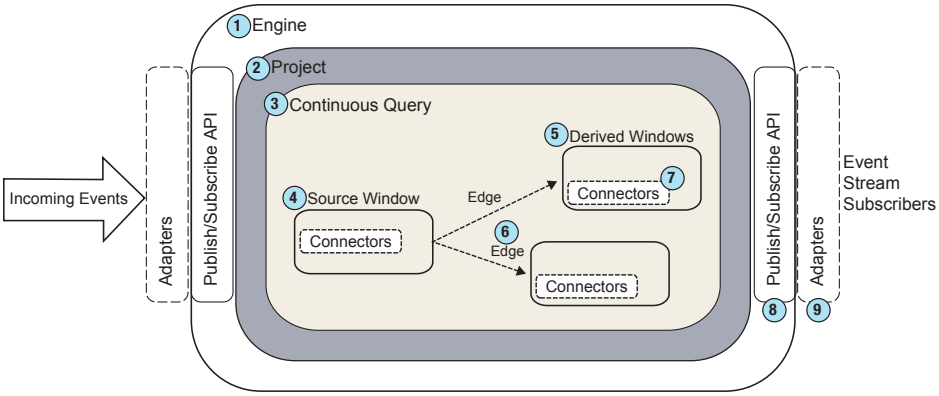
- What event streams are published into an application, and with what protocol and format?
- What happens to the data? That is, how are event streams transformed and analyzed?
- What are the resulting event streams of interest? What applications subscribe these event streams, and in what format and protocol?

Your answers to these questions determine the structure of your 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.

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 XML 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.

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

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.

Opcode	Description
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.

Conversion of Events into Binary Code

When events are published into source windows, they are converted into binary code with fast field pointers and control information. This conversion improves throughput performance.

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.

For CSV conversion to binary, refer to the example application "csv2bin" under the **`examples/cxx`** directory of the SAS Event Stream Processing installation. The readme.examples file in **`$DFESP_HOME/examples`** explains how to use this example in order to convert CSV files to event stream processor binary files. The example shows you how to perform the conversion in C++ using methods of the C client API. You can also convert using the Java or Python client API.

The following code example reads in binary events from `stdin` and injects the events into a running project

Note: Events for only one window can exist in a given file. For example, all the events must be for the same source window. It also groups the data into blocks of 64 input events to reduce overhead, without introducing too much additional latency.

```
// For windows it is essential that you read binary
// data in BINARY mode.
//
dfESPfileUtils::setBinaryMode(stdin);
```

```
   // Trade event blocks are in binary form and
   // are coming using stdin.
      while (true)
{
   // more trades exist
   // Create event block.
      ib = dfESPeventblock::newEventBlock(stdin,
           trades->getSchema());
       if (feof(stdin))
           break;
       sub_project->injectData(subscribeServer,
                               trades, ib);
}
sub_project->quiesce(); // Wait for all input events to be processed.
```

Understanding Event Blocks

Event blocks contain zero or more binary events, and 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.

Event blocks can be transactional or normal.

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 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 they 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 event block ID.

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 using the C++ modeling layer

SAS Event Stream Processing provides three modeling APIs to implement engines:

- The XML Layer enables you to define single engine definitions and to define an engine with dynamic project creations and deletions. You can use the XML Layer with other products such as SAS Visual Scenario Designer to perform visual modeling.
- You can use SAS Event Stream Processing Studio to create event stream processing models and generate XML code based on those models.

- 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.

In the XML Layer, the XML server is an engine process that accepts event stream processing definitions for projects and engines. Because the XML server is an instantiated engine, it ignores any engine specification that you submit to it. Instead, it instantiates the project within the submitted engine.

You can embed event stream processing engines within application processes through the C++ Modeling API. The application process that contains the engine can be a server shell, or it can be a working application thread that interacts with the engine threads.

Deciding whether to implement multiple projects or multiple continuous queries depends on your processing needs. For the XML 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
- 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++ or DS2.

Input handlers written in DS2 can use features of the SAS Threaded Kernel library so that you can run existing SAS models in a procedural window. You can do this only when the existing model is additive in nature and can process one event at a time.

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. A project can specify the level of determinism for incremental computations. It 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 specified or 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 a *source window* and one or more *derived windows*. Windows are connected by *edges*, which have an associated direction.

SAS Event Stream Processing supports the following window types:

Window Type	Description
source window	Specifies a source window of a continuous query. 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.
compute window	Defines a compute window, which enables a one-to-one transformation of input events into output events through the computational manipulation of the input event stream fields.

Window Type	Description
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.
aggregate window	An aggregate window is 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.
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	Specifies a window with 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.
join window	Takes two input windows and a join type. A join window 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.
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 "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).
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.

Window Type	Description
union window	Specifies a simple join that merges one or more streams with the same schema. All input windows to a union window must have the same schema. The default value of the strict flag is **true**, which means that the key merge from each window must semantically merge cleanly. In that case, you cannot send an Insert event for the same key using two separate input windows of the union window. When the strict flag is set to **false**, it loosens the union criteria by replacing all incoming Inserts with Upserts. All incoming Deletes are replaced with safe Deletes. In that case, deletes of a non-existent key fail without generating an error.

Renewing Your Software License

A valid license file is required to run any applications that use SAS Event Stream Processing.

Your Software License Renewal Confirmation message contains a license file. Copy that file to **$DFESP_HOME/etc/license**.

Note: For SAS Event Stream Processing 4.2, **$DFESP_HOME/etc/license** is **/opt/sas/viya/home/SASEventStreamProcessingEngine/4.2.0/etc/license**.

If an application changed the license location using the C++ Modeling API `dfESPlibrary::Initialize`, copy the license file to that location.

Writing an Event Stream Processing Application

1. Create an engine and instantiate it using XML server or within a C++ application.

   For more information about the XML server, see "Using the XML Server" in *SAS Event Stream Processing: Using the XML Layer*.

   For more information about the C++ key modeling objects to use to write an application, see "C++ Modeling Objects for SAS Event Stream Processing " in *SAS Event Stream Processing: Programming Reference*.

2. Publish one or more event streams into the engine using the publish/subscribe API, connectors, adapters, SAS Event Stream Processing Studio, Streamviewer, or by using the `dfESPcontquery::injectEventBlock()` method for C++ models.

   You can publish and subscribe in one of the following ways:

   - through the Java, C, or Python publish/subscribe API
   - through the packaged connectors (in-process classes) or adapters (networked executables) that use a publish/subscribe API, SAS Event Stream Processing Studio, or Streamviewer
   - using the in-process callbacks for subscribe or the inject event block method of continuous queries

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

   Connectors are C++ classes that are instantiated in the same process space as the event stream processor. Connectors can be used from 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 “Using Connectors and Adapters” in *SAS Event Stream Processing: Connectors and Adapters*.

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

When you start a 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.