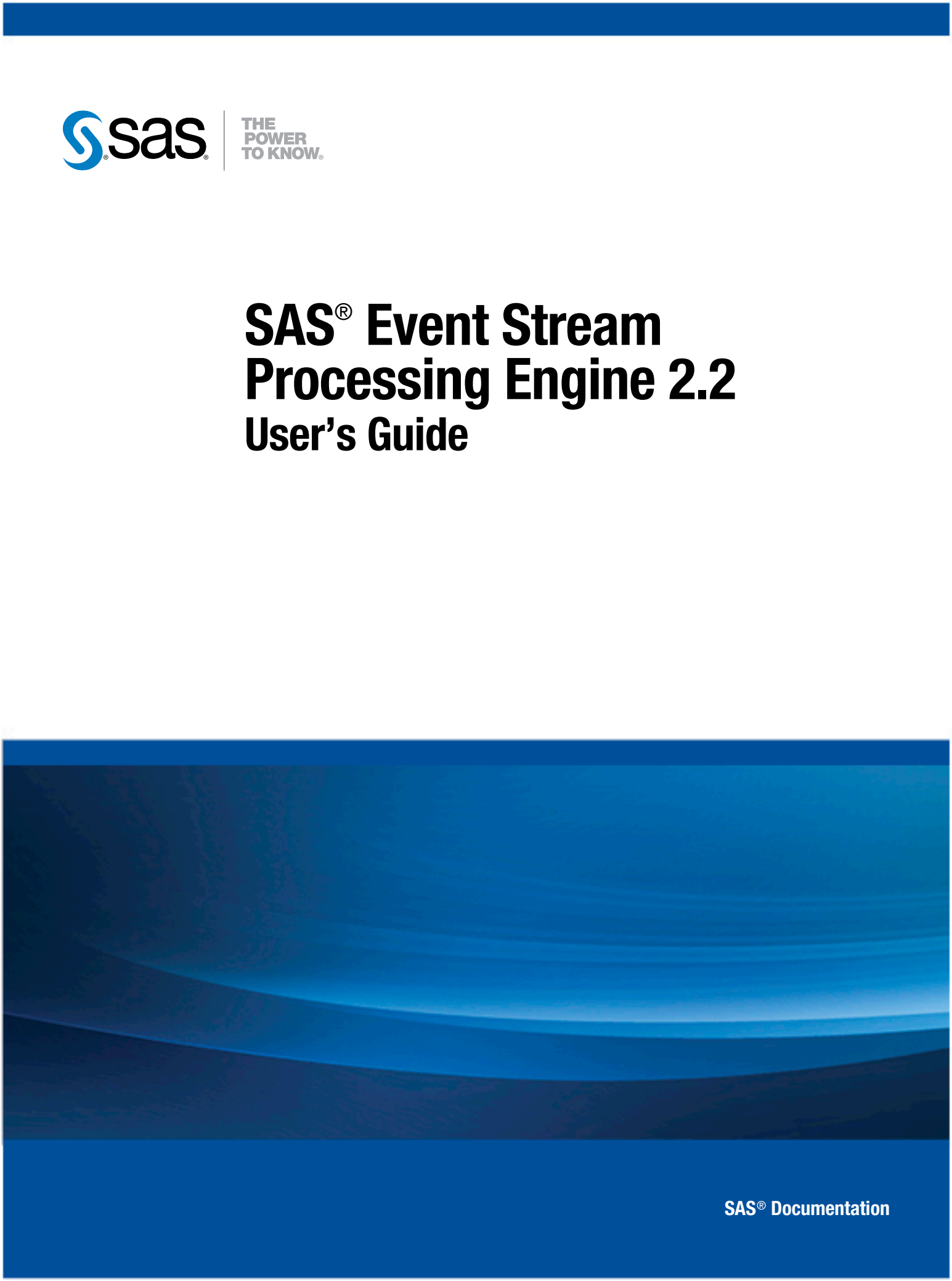


SAS® Event Stream Processing Engine 2.2
User's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2014. *SAS® Event Stream Processing Engine 2.2: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Event Stream Processing Engine 2.2: User's Guide

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

September 2014

Contents

About This Book

Audience

This document provides information for programmers to use SAS Event Stream Processing Engine. It assumes knowledge of object-oriented programming terminology and a solid understanding of object-oriented programming principles. It also assumes a working knowledge of the SQL programming language and of relational database principles. Use this document with the application programming interface (API) HTML documentation that is shipped with the product.

What's New in SAS Event Stream Processing Engine

Overview

The September 2014 release of SAS Event Stream Processing Engine 2.2 provides the following enhancements:

- improved protocol buffer conversion to event blocks
- revised connectors.excluded file
- new publish/subscribe command version
- enhanced functionality for join expression for outer joins
- new LASR JAR file
- restoration of reference-counted data
- improved handling of redundant updates
- memory optimization
- better memory allocation
- new function available through the Publish/Subscribe APIs

SAS Event Stream Processing Engine 2.2 provides the following new functionality.

- redesigned and enhanced XML Modeling Layer
- new and improved adapters
- enhancements to the Publish/Subscribe API
- new and enhanced aggregate functions
- support for SAS Visual Scenario Designer
- integration with SAS BI Dashboard
- performance and optimization improvements
- new expression engine initialization functions
- improved error-handling semantics
- new window capabilities
- new engine-level specification to handle modeling errors

Improved Protocol Buffer Conversion to Event Blocks

The conversion of a protocol buffer to an event block has been improved. It now handles the conversion when the following occurs:

- the protocol buffer message contained repeated fields of type **`message`**
- at least one of those message fields contained a repeated message field

Revised Connectors.excluded File

The entry **`?tdata?`** was added to the Connectors.Excluded file to prevent default loading of the Teradata connector during engine boot.

New Publish/Subscribe Command Version

The publish/subscribe command version was incremented to account for the new **`??action=quiesce?`** URL parameter. This is relevant in cases where either the publish/subscribe server or the publish/subscribe client is a version other than SAS Event Stream Processing Engine 2.2. A warning message states that the versions are different but not incompatible.

Enhanced Functionality for Join Expression for Outer Joins

Outer joins that use calculation expression for non-key fields now correctly use null for the field values on the non-matching side of the outer join.

New LASR JAR File

SAS Event Stream Processing Engine now includes **`lib/sas.lasr-1.0.jar`**.

Restoration of Reference-Counted Data

Data reference counts are now maintained when a model or project is restored from persisted state.

Improved Handling of Updates That Do Not Change an Event

Handling of updates that do not change an event has been optimized so that data is not duplicated. The updates are caught earlier in the computational cycle, providing more computational efficiency.

Memory Optimization

Savings of 24 bytes per event have been made for normal event processing. An increase of 8 bytes per event occur for events that measure system latency.

Aggregate windows have also been optimized, saving a significant amount of memory per group. The performance of aggregate windows with very high cardinality group-by keys has been improved.

Better Memory Allocation

The default memory allocator for Linux has an exceptionally high thread affinity for memory arena localization. A thread that allocates a lot of memory and that is later freed from another thread remains largely unused, sitting idle in the application heap. This lack of use can result in exceptionally high memory utilization for the process.

Using TMmalloc on 64–bit Linux systems, the thread to memory arena affinity is now more balanced. Memory allocated from one thread and freed in another becomes reusable across a wider number of other threads.

New Function Available through the Publish/ Subscribe APIs

The maxQueueSize function configures the maximum size of the queues that are used to enqueue event blocks sent to subscribers in a project, query, or window. It also specifies the behavior when the maximum size is reached. The function is available through the C and the Java Publish/Subscribe APIs.

For more information, see "Using the Publish/Subscribe API" on page 109.

Redesigned and Enhanced XML Modeling Layer

The XML modeling layer has been significantly redesigned and enhanced. It now aligns more closely with the functionality and object model of the C++ Modeling API.

Two new commands were added:

- **`dfesp_xml_client`** can be used to send control commands to the XML server
- **`dfesp_xml_validate`** can be used to validate that an XML model is syntactically valid before submitting it to the XML server

For more information, see Chapter 4, "Using the XML Modeling Layer," on page 39.

New and Improved Adapters

SAS Event Stream Processing Engine provides the following new adapters:

- An adapter that can subscribe to event streams for periodic LASR appends. This is the preferred adapter for the SAS LASR Analytic Server when low latency is the priority. This adapter uses the LASR Java API.

  To run this adapter in a UNIX environment requires the installation of an SSH client. Do not run this adapter in a Microsoft Windows environment.

  For more information, see "Using the SAS LASR Analytic Server Adapter" on page 162.

- A Hadoop HDAT adapter that you can use when the priority is high throughput rather than low latency. This adapter uses the proprietary SAS HDAT format. It can subscribe to event streams and periodically append them to LASR. For more information, see "Using the File and Socket Adapter" on page 163.

- A Teradata adapter that publishes and subscribes. This adapter replaces the ODBC-based adapter that was provided in previous releases. The new adapter performs parallel loads to and data queries from the Teradata Data Fabric. For more information, see "Using the Database Adapter" on page 165.

The PI Adapter has been completely rewritten to use PI's Asset Framework API. The adapter now enables both publish and subscribe operations. For more information, see "Using the PI Adapter" on page 177.

Enhancements to the Publish/Subscribe API

The publish/subscribe API now provides the following enhancements:

- You can now specify in the publish/subscribe URL that you do not want to receive retention deletes for a subscribe. This can be useful for subscribes from SAS BI Dashboard or from any visualization subscriber. For more information, see "Using the Publish/Subscribe API" on page 109.

- The C++ Publish/Subscribe API, the Java Publish/Subscribe API, and relevant connectors and adapters now support Google Protocol Buffers. For more information, see "Publish/Subscribe API Support for Google Protocol Buffers" on page 227.

New and Enhanced Aggregate Functions

SAS Event Stream Processing Engine 2.2 now provides the following aggregate functions:

- **ESP_aCountNull**
- **ESP_aCountNonNull**
- **ESP_aCountDistinct**

It provides enhancements to the following aggregate functions:

- **ESP_aLast**
- **ESP_aFirst**
- **ESP_aMax**
- **ESP_aMin**

For more information, see "Using Aggregate Functions" on page 216.

Support for SAS Visual Scenario Designer

SAS Event Streams Processing Engine now supports XML models generated by SAS Visual Scenario Designer. SAS Visual Scenario Designer must be ordered separately.

SAS BI Dashboard Provider

A new data provider was added in SAS Event Streams Processing Engine 2.2 that interfaces with SAS BI Dashboard for continuous visualization of event streams. With this data provider, SAS BI Dashboard can provide one-second updates to event stream visualizations.

For more information, see Chapter 9, "Visualizing Event Streams," on page 105.

Performance and Optimization Improvements

Improvements were made to the modeling and publish/subscribe APIs to enhance performance and memory usage:

- The Java publish/subscribe API was improved to optimize memory usage.

- Procedural windows with handlers written in DS2 have been optimized with respect to speed and memory usage.
- Aggregation functions have been optimized for better performance.

New Expression Engine Initialization Functions

The following functions have new functionality or are new for SAS Event Stream Processing Engine 2.2:

- **`dfESPwindow::setSplitter()`** was enhanced to permit an optional initialization expression for a window's optional output splitter.
- **`dfESPexpression_window::expEngInitializeExp()`** was added to enable the initialization of an expression window's expression engine.

These functions can be used to define expression engine variables that can be used in expression strings. For more information, see Chapter 2, "Using Expressions," on page 9.

Improved Error-Handling Semantics

SAS Event Stream Processing Engine 2.2 changes the computational semantics of data flow. In order to make the engine more resilient in handling run-time errors in projects, a uniform error-handling semantic has been introduced.

You can think of the computational step for any window as consisting of two parts:

- computation of output events from the input events
- application of the computed output events to the window index in order to adjust state

For derived windows, an error during either of these steps results in a fatal run-time error. Depending on the engine setting, such a fatal error can cause either the engine to stop all processing within the project or to terminate the process entirely. For source windows, an error at either of these steps is not as severe. A source block that fails to enter the system cannot introduce computational inconsistencies.

Regardless of the disposition of the engine fatal error, the input events that cause the error are not applied to the project's data flow model. The events that cause the problem are logged through the engine's error-handling mechanism.

For more information, see the Details of "dfESPengine" on page 14.

New Window Capabilities

SAS Event Stream Processing Engine 2.2 provides the following new window capabilities.

- A new window type, **`textContext`**, is available. You can use this window type to analyze a string field from an event's input to parse text and extract concepts,

entities, and facts. You can also categorize extracted terms and phrases. Events generated from text analysis can be analyzed by other window types.

- A compute window now enables you to specify different key fields from that of its parent.
- A new context mechanism in compute windows supports sharing of data between field expressions.
- Join, procedural, and source windows can be stateless and not insert only. This enables pass through models where data is joined with aggregated values without being retained as it passes through the system.
- Full outer joins are now supported between two fact tables.

For more information, see Chapter 3, "Programming with the C++ Modeling API," on page 13.

New Engine-Level Specification to Handle Modeling Errors

In previous releases, the event stream processing engine logged major errors and halted. This created core files for potential debugging. SAS Event Stream Processing Engine 2.2 enables you to specify how to handle major modeling errors for each project. These options are as follows:

- Abort the engine with a core file.
- Exit the engine without a core file.
- Exit the project, not the engine.

For information about how to do this through the C++ Modeling API, see "dfESPengine" on page 14. For information about how to do this through the XML Modeling Layer, see "Using the XML Modeling Layer" on page 39.

Recommended Reading

SAS Event Stream Processing Engine is supported by the following documents:

- *SAS Event Steam Processing Engine: Overview* provides an introduction to the product and an illustrative example.
- *SAS Event Stream Processing Engine: User's Guide* describes the product and provides technical details for writing event stream processing applications.
- Open **`$DFESP_HOME/doc/html/index.html`** in a web browser to access detailed class and method documentation for the C++ modeling, C, and Java™ client publish/subscribe APIs. The documentation is organized by modules, namespaces, and classes.

  Specifically, documentation about the following topics is provided:

  - C++ Modeling API
    - SAS Event Stream Processing API
    - SAS Event Stream Processing Connector API
  - Publish/Subscribe API
    - SAS Event Stream Processing Publish/Subscribe C API
    - SAS Event Stream Processing Publish/Subscribe Java API

For a complete list of SAS books, go to support.sas.com/bookstore. If you have questions about which titles you need, please contact a SAS Book Sales Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-3228
Fax: 1-919-677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/bookstore

Chapter 1

Overview to SAS Event Stream Processing Engine

Product Overview

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

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

Event stream processing applications typically perform real-time analytics on event streams. 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:

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

- cyber security analytics

Event stream processing enables the user to analyze continuously flowing data over long periods of time where low latency incremental results are important. Event stream processing applications can analyze millions of events per second, with latencies in the milliseconds.

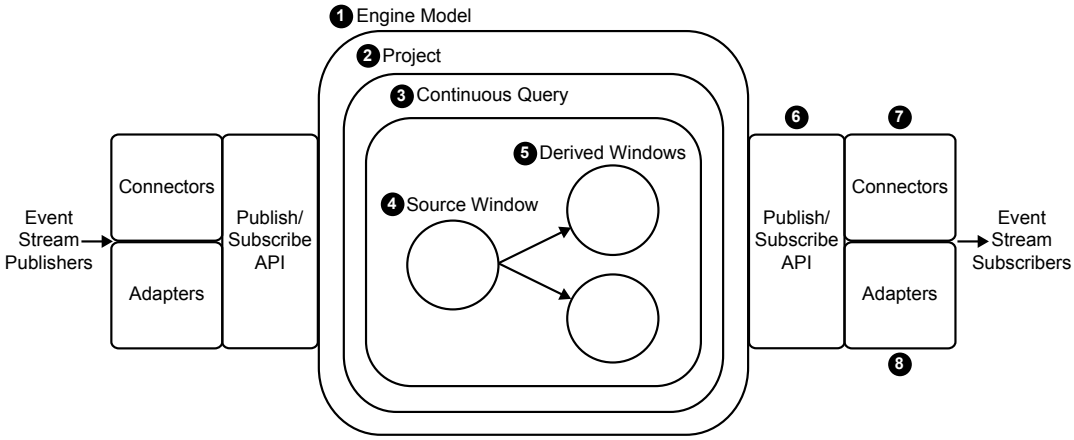
Conceptual Overview

SAS Event Stream Processing Engine enables a programmer to write event stream processing applications that continuously analyze events in motion. When designing an application, programmers must answer the following questions:

- What is the model that tells the application how to behave?
- How are event streams (data) to be published into the application?
- What transformations must occur to those event streams?
- How are the transformed event streams to be consumed?

A *model* is a user specification of how input event streams from publishers are transformed into meaningful output event streams consumed by subscribers. The following figure depicts the model hierarchy.

Figure 1.1 *Instantiation of an Engine Model*



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

2 The engine contains one or more *projects*, each uniquely named. Projects contain dedicated *thread pools* that are specified relative to size. 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, which is a set of connected nodes that follow a direction down one or more parallel paths. Continuous queries contain data flows, which are data transformations of incoming event streams.

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

5 Source windows are connected to one or more *derived windows*. Derived windows can be connected to other derived windows.

6 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.

7 *Connectors* use the publish/subscribe API to publish or subscribe event streams to and from an engine. Connectors bypass sockets for a lower-level inject call because they are in process to the engine.

8 *Adapters* are stand-alone executable programs that can be networked. Adapters also use the publish/subscribe API.

Implementing Engine Models

SAS Event Stream Processing Engine provides two modeling APIs to implement event stream processing engine 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.
- The XML Modeling Layer enables you to define single engine definitions that run like those implemented through the C++ Modeling API. You can also use it to define an engine with dynamic project creations and deletions. Also, you can use it with products such as SAS Visual Scenario Designer to perform visual modeling.

Event stream processing engine models can be embedded within application processes through the C++ Modeling API or can be XML factory servers. 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.

The XML factory server is an engine process that accepts event stream processing definitions in one of two ways:

- in the form of a single, entire engine definition
- as create or destroy definitions within a project, which can be used to manipulate new project instantiations in an XML factory server

For either modeling layer, the decision to implement multiple projects or multiple continuous queries depends on your processing needs. For the XML factory server, multiple projects can be dynamically introduced and destroyed because the layer is being used as a service. For both modeling layers, multiple projects can be used for modularity 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
- a mixed threading model to manipulate both

Because continuous queries can also be used 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, but loop-back conditions are not permitted. Event streams must be published or injected into source windows through the publish/subscribe API or through the continuous query inject method.

Within a continuous query, you can implement relational, rule-based, and procedural operators on derived windows. The relational operators include the following SQL primitives: join, copy, compute, aggregate, filter, and union. The rule-based operators perform pattern matching and enable you to define temporal event patterns. The procedural operators enable you to write event stream input handlers in 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.

Various connectors and adapters are provided with SAS Event Stream Processing Engine, but you can write your own connector or adapter using the publish/subscribe API. Inside model definitions, you can define connector objects that can publish into source windows or that can subscribe to any window type.

Understanding Continuous Queries

Within a continuous query, windows can transform or analyze data, detect patterns, or perform computations. Continuous query processing follows these steps:

1. An event block (with or without atomic properties) containing one or more events is injected into a source window.
2. The event block flows to any derived window 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 that 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 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 resultant events for that event block. Therefore, it has nothing to pass to the next set of connected derived windows.

   For information about the C++ programming objects available with the SAS Event Stream Processing Engine that you use to implement windows, see Chapter 3, “Programming with the C++ Modeling API,” on page 13. For information about the XML code you can use to implement windows, see Chapter 4, “Using the XML Modeling Layer,” on page 39.

Understanding Events

An event is a packet of data that is accessible as a collection of fields. One or more fields of the event must be designated as a primary key. Key fields enable the support of operation codes (opcodes) to process data within windows. The opcodes supported by SAS Event Stream Processing Engine consist of Delete, Insert, Update, and Upsert.

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.

When programming, if 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.

The SAS Event Stream Processing Engine has a publish/subscribe API. In-process connectors and out-of-process adapters use this API to either publish event streams into source windows or subscribe to any window's output event stream. This API is available in Java or C.

Using connectors or adapters, you can publish or subscribe to events in many formats and from various systems. Example formats include CSV, JSON, binary, and XML. Example publishing and subscribing systems include databases, financial market feeds, and memory buses. You can also write your own connectors or adapters that use the publish/subscribe API.

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

Understanding Event Blocks

Event blocks contain zero or more binary events, and publish/subcribe clients send and receive event blocks to or from the SAS Event Stream Processing Engine. Because publish/subscribe carries 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	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 a bad records file, which can be processed further.
Normal	Is not atomic. Events are packaged together for efficiency, but are individually treated once they are injected into a source window.

Events persist in their 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.

Getting Started with SAS Event Stream Processing Engine

Installing and Configuring SAS Event Stream Processing Engine

Instructions to install and configure SAS Event Stream Processing Engine are provided in a ReadMe file available in your software depot.

Note: Do not overlay SAS Event Stream Processing Engine 2.2 over a previous product release. If you want to preserve files from the previous release, rename the associated directories.

Using the SAS Event Stream Processing Engine

After you install SAS Event Stream Processing Engine, you write event stream processing models using the C++ Modeling API or the XML modeling layer and then execute them. Sample models in both C++ and XML are available in the **`src`** directory of the installation. A file named ReadMe.examples in the **`src`** directory describes each example and how to run it.

You need a valid license file in order to run any applications using SAS Event Stream Processing Engine. License files are ordinarily stored in **`etc/license`**. If you do not have a license file, please contact your SAS representative.

Note: If you store the license file in a different location from **`etc/license`**, you need to modify the sample applications and change the calls to **`dfESPlibrary::Initialize`**. For more information, see the API documentation available in **`$DFESP_HOME/doc/html`**.

Writing an Application with SAS Event Stream Processing Engine

An engine model specifies how to process event streams published into an engine. These models define data transformations, known as continuous queries, that are performed on the events of one or more event streams. An event stream processing application instantiates a model with one or more dedicated thread pools. These thread pools are defined within projects.

Follow these steps to write an event stream processing application:

1. Create an engine model and instantiate it within an application or by the XML factory server.
2. Publish one or more event streams into the engine using the publish/subscribe API, connectors, adapters, or by using the **`dfESPcontquery::injectEventBlock()`** method for C++ models.
3. Subscribe to relevant window event streams within continuous queries using the publish/subscribe API,, connectors, adapters, or using the **`dfESPwindow::addSubscriberCallback()`** method for C++ models.

You can use the C++ Modeling API or the XML Modeling Layer to write event stream processing application models. For more information about the C++ key modeling objects to use to write an application, see Chapter 3, “Programming with the C++ Modeling API,” on page 13.

The XML Modeling Layer uses a factory server to instantiate and execute event stream process modeling objects that are defined in an XML file. For more information, see Chapter 4, “Using the XML Modeling Layer,” on page 39.

You can publish and subscribe one of three ways:

- through the Java or C publish/subscribe API
- through the packaged connectors (in-process classes) or adapters (networked executables) that use the publish/subscribe API
- 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 Chapter 10, “Using the Publish/Subscribe API,” on page 109.

Connectors are C++ classes that are instantiated in the same process space as the event stream processor. Connectors can be used from within C++ models as well as XML models. For more information, see Chapter 11, “Using Connectors,” on page 129.

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 Chapter 12, “Using Adapters,” on page 161.

Chapter 2

Using Expressions

Overview to Expressions

Event stream processing applications can use expressions to define the following:

- filter conditions in filter windows
- non-key field calculations in compute, aggregate, and join windows
- matches to window patterns in events of interest
- window-output splitter-slot calculations (for example, use an expression to evaluate where to send a generated event)

You can use user-defined functions instead of expressions in all of these cases except for pattern matching. With pattern matching, you must use expressions.

Writing and registering expressions with their respective windows can be easier than writing the equivalent user-defined functions. Expressions run more slowly than functions. For very low-latency applications, you can use user-defined functions to minimize the overhead of expression parsing and processing.

Use prototype expressions whenever possible. Based on results, optimize them as necessary or exchange them for functions. Most applications use expressions instead of functions, but you can use functions when faster performance is critical.

For information about how to specify DataFlux expressions, refer to the *DataFlux Expression Language: Reference Guide*. The SAS Event Stream Processing Engine uses a subset of the documented functionality, but this subset is robust for the needs of event stream processing.

Expression engine instances run window and splitter expressions for each event that is processed by the window. You can initialize expression engines before they are used by expression windows or window splitters (that is, before any events stream into those windows). Each expression window and window splitter has its own expression engine instance. Expression engine initialization can be useful to declare and initialize expression engine variables used in expression window or window splitter expressions. They can also be useful to declare regular expressions used in expressions.

Initialize expression window expression engines using **dfESPexpression_window::expEngInitializeExp()**. Initialize window splitter expression engines using **dfESPwindow::setSplitter()**. You can find example uses of both of these in the event stream processing installs in **$DFESP_HOME/src**. The window splitter example is named regex, and the expression window example is named splitter_with_initexp.

Data Type Mappings

A one-to-one mapping does not exist between the data types supported by the SAS Event Stream Processing Engine API and those supported by the DataFlux Expression Engine Language. The following table shows the data type mappings.

Table 2.1 *Expression Data Type Mappings Table*

Notes and Restrictions	DataFlux Expressions	Event Stream Processing Engine Expressions
None	String (utf8)	String (utf8)
Seconds granularity	date (second granularity)	date (second granularity)
Constant milliseconds in dfExpressions not supported	date (second granularity)	timestamp (microsecond granularity)
64-bit conversion for dfExpressions	Integer (64 bit)	Int32 (32 bit)
64-bit, no conversion	Integer (64 bit)	Int64 (64 bit)
real 192-bit fixed point, double 64-bit float	real (192 bit fixed decimal)	double (64 bit IEEE)
192-bit fixed point, no conversion	real (192 bit fixed decimal)	money (192 bit fixed decimal)

Using Blue Fusion Functions

Event stream processing expressions support the use of the DataFlux Data Management Platform quality functions (Blue Fusion Functions). The following functions are fully documented in the *DataFlux Expression Language: Reference Guide*:

- bluefusion.case
- bluefusion.gender
- bluefusion.getlasterror
- bluefusion.identify
- bluefusion_initialize
- bluefusion.loadqkb
- bluefusion.matchcode
- bluefusion.matchscore
- bluefusion.pattern
- bluefusion.standardize

To use these functions, you must separately order and download the SAS DataFlux QKB (Quality Knowledge Base). You must set two environment variables as follows:

- **`DFESP_QKB`** to the share folder under the SAS DataFlux QKB installation. When you have installed SAS Event Stream Processing Engine on Linux systems, this share folder is **`/`*`QKB_root`*`/data/ci/`*`esp_version_number_no_dots`* (for example, **`/QKB/data/ci/22`**).
- **`DFESP_QKB_LIC`** to the full file pathname of the SAS DataFlux QKB license.

After you set up SAS DataFlux QKB for SAS Event Stream Processing Engine, you can include these functions in any of your SAS event stream processing expressions. These functions are typically used to normalize event fields in the non-key field calculation expressions in a compute window.

For an example, see Appendix 1, “Example: Using Blue Fusion Functions,” on page 231.

Chapter 3

Programming with the C++ Modeling API

Overview to the C++ Modeling API

The C++ Modeling API provides a set of classes with member functions. These functions enable you to embed an engine with dedicated thread pools into an application's process space. An application can define and start an engine, which would make it an event stream processing server.

Alternatively, you can embed an engine into the process space of an existing or a new application. In that case, the main application thread is focused on its own chores. It interacts with the embedded engine as needed.

The following sections explain how to use the C++ key modeling objects. For information about how to define models using the XML Modeling Layer, see Chapter 4, "Using the XML Modeling Layer," on page 39.

Dictionary

dfESPengine

specifies the top-level container or manager in a model. Engines typically contain one or more projects. Create only one engine instance, which instantiates fundamental services such as licensing, logging, and expression handling. An attempt to create a second engine instance returns the pointer to the first created engine instance.

Syntax

dfESPengine **engine_name*;
engine_name= dfESPengine::initialize(argc, argv, "*engine_name*",
pubsub_ENABLE(*port#*) | pubsub_DISABLE,
<*loglevel*>, <*logConfigFile*>);

Required Arguments

argc
argument count as passed into main

argv
argument vector as passed into main — accepts **`-t`** *textfile.name* to write output and **`-b`***badevent.name* to write events that failed to be applied to a window index

engine_name
user-supplied name of the engine

pubsub_ENABLE(*port#*) | pubsub_DISABLE
indicate whether to enable (on a user-specified *port#*) or disable publish/subscribe

Optional Arguments

logLevel
the lower threshold for displayed log messages. The default value is **`dfESPLLTrace`**.

logConfigFile
a logging facility configuration file. The default is to configure logging to go to standard output.

licKeyFile
a fully qualified pathname to a license file. The default is **`$DFESP_HOME/etc/license/esp.lic`**.

Details

You can use the following method to tell an engine how to handle fatal errors that occur in a project.

static DFESP_API void dfESPengine::setProjectFatalDisposition (projectFatal_t *dispositionFlag*)

Set the *dispositionFlag* to one of the following values:

- 0 — exit with the engine process
- 1 — exit and generate a core file for debugging, or stop all processing
- 2 — disconnect publish/subscribe, clean up all threads and memory, and remove the process from the engine while leaving the engine up and processing other projects

Example

The following example creates, starts, stops, and shuts down an engine.

```
// Create the engine container.
dfESPengine *engine;
engine = dfESPengine::initialize(argc, argv, "engine", pubsub_DISABLE);

// Create the rest of the model and then start the projects.
//     Creating the rest of the model is covered in the
//     subsequent sections.
engine->startProjects();

// The project is running in the background and you can begin
//     publishing event streams into the project for execution.

/* Now cleanup and shutdown gracefully */

// First, stop the projects.
engine->stopProjects();

// Next, shutdown the engine and its services (this frees all
//     the modeling resources from the engine down through
//     the windows).
engine->shutdown();
```

dfESPproject

specifies a container that holds one or more continuous queries and are backed by a thread pool of user-defined size

Syntax

dfESPproject **projectname*
= *engine_name*—>newProject(“*project_label*”);

Required Arguments

projectname
user-supplied name of the project

engine_name
user-supplied name of the engine as specified through **`dfESPengine`**.

project_label
user-supplied description of the project

Details

A project can specify the level of determinism for incremental computations. The data flow model is always computationally deterministic. Intermediate calculations can occur at different times across different runs of a project when it is multi-threaded. 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 SAS Event Stream Processing Engine model, each window always processes all data in order. Therefore, data received by a window is never rearranged and processed out of order.

The levels of determinism supported by a project are as follows:

- full concurrency (default) - data received by a window is processed as soon as it is received and forwarded on to any dependent window. This is the highest performing mode of computation. In this mode, a project can use any number of threads as specified by the **`setNumberOfThreads(max thread)`** method.
- tagged token - implements single-transaction in, single-transaction out semantics at each node of the model. In this mode, a window imposes a diamond patter, splitting the output and then rejoining the split paths together. It merges outputs (per unique transaction) from each path into a single transaction. A single transaction in the top of the diamond produces a single output at the bottom.

The **`newProject()`** method for the dfESPengine class takes a final parameter (**`true`** | **`false`**) that indicates whether tagged token data flow should be enabled. If you do not specify this final parameter, the value defaults to false.

- Thus, to specify full concurrency:

  ```
  dfESPproject *project = engine->newProject("MyProject”);
  ```

  or

  ```
  dfESPproject *project = engine->newProject("MyProject”, false);
  ```

- And to specify tagged token:

  ```
  dfESPproject *project = engine->newProject("MyProject”, true);
  ```

For easier debugging and full consistency in output for testing, run with tagged token **`true`**. Set the number of threads in a project to 1. This is the slowest way to run a project. Nonetheless, as long as you are using time-based retention policies, you can be assured that the output is consistent from run to run.

Example

The following code fragment shows how to create a project, add a memory store, set the thread pool size, and add continuous queries. It is run with a level of determinism of full consistency.

```
// Create the project containers.
dfESPproject *project = engine->newProject("MyProject”);

// Create a memory depot for the project to handle the generation
//     of primary indices and event storage;

dfESPeventdepot_mem* depot;
depot = project->newEventdepot_mem("memDepot_01");
```

```
project->setNumThreads(3);  //set the thread pool size for project.

// After you have started the projects using the startProjects()
//     method shown  in the dfESPengine section above, then you
//     can publish or use dfESPproject::injectData() to inject
//     event blocks into the source windows. You can also use
//     the dfESPproject::quiesce() method to block until all
//     of the data in the continuous queries of the project are
//     quiesced. You might also want to do this before stopping
//     the projects.

project->quiesce();
project->stopProject();
```

dfESPeventdepot

creates a high-level factory object that builds and tracks primary indexes for all the windows in a project that use it.

Syntax

dfESPeventdepot_mem* *depot_name*;
depot_name =*projectname*—>newEventdepot_mem("*depot_label*");

Required Arguments

depot_name
user-supplied name of the depot object

project_name
user-supplied name of the project as specified in **`dfESPproject`**

depot_label
user-supplied description of the depot object

Example

The only concrete implementation of the **`dfESPeventdepot`** class is a memory-based storage and indexing mechanism encapsulated in the **`dfESPeventdepot`** class. When writing models, you typically create a single instance of **`dfESPeventdepot`** for each project and use it to manage the indexing of events for all windows of a project:

```
dfESPeventdepot_mem *edm = project->newEventdepot_mem("myDepot");
```

dfESPcontquery

specify a continuous query object. This is a container that holds a collection of windows and enables you to specify the connectivity between windows.

Syntax

dfESPcontquery *_query_name_
_query_name_ = _projectname_—> newContQuery("_query_label_");

Required Arguments

query_name
user-supplied name of the query object

projectname
user-supplied name of the project, as specified in **`dfESPproject`**

query_label
user-supplied description of the query

Example

Suppose that there are two windows, **`swA`** and **`swB`**, that are joined to form window **`jwC`**. Window **`jwC`** is aggregated into window **`awD`**. Build the continuous query as follows, using the **`addEdge`** function:

```
dfESPcontquery *cq;
cq = project->newContquery("continuous query #1");

cq->addEdge(swA, jwC); // swA --> jwC
cq->addEdge(swB, jwC); // swB --> jwC
cq->addEdge(jwC, awD); // jwC --> awD
```

This fully specifies the continuous query with window connectivity, which is a directed graph.

dfESPwindow_source

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.

Syntax

dfESPwindow_source *_windowID_;
windowID=_contquery_name_—> newWindow_Source
("_window_label_", _depot_name_, dfESPIndextypes::_index_,
schema);

Required Arguments

windowID
user-supplied identifier of the source window

contquery_name
user-supplied name of the continuous query object specified in **`dfESPcontquery`**

window_label
user-supplied description of the window

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

schema
user-supplied name of the schema as specified by **`dfESPstring`**.

Example

Here is an example of how to specify a source window:

```
dfESPwindow_source *sw;
sw = cq->newWindow_source("mySourceWindow", edm,
     dfESPindextypes::pi_HASH, sch);
```

Before creating this source window, you could use **`dfESPstring`** to specify a schema. For example

```
dfESPstring sch = dfESPstring("ID*:int32,symbol:string,price:double");
```

Alternatively, you could specify the **`dfESPstring`** definition directly into the newWindow schema field when you define the window type.

You can set event state retention for source windows and copy windows 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 automatically by the engine when they exceed the window's retention policy.

Set the retention type on a window with the **`setRetentionParms()`** call. You can set type by count or time, and as either jumping or sliding. For more information, see "Retention Policies for Fully Stateful Indexes" on page 214.

dfESPwindow_filter

specifies a filter window, which is a computational window with a registered Boolean filter function or expression.

Syntax

dfESPwindow_filter **windowID*;
windowID=*query_name*—> newWindow_filter("*window_label*",
depot_name, dfESPIndextypes::*index*;
windowID->setFilter(*filterFunction* | *filterExpression*);

Required Arguments

windowID
user-supplied ID of the filter window

query_name
user-supplied name of the query object specified in **`dfESPcontquery`**

window_label
user-supplied description of the window

depot_name
user-supplied name of the depot as specified by **dfESPeventdepot**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

filterFunction
user-supplied identifier of the filter function

filterExpression
user-supplied identifier of the filter expression

Details

The filter function or expression, which is set by the **setFilter** method, is called each time that a new event block arrives in the filter window. The filter function or expression uses the fields of the events that arrive to determine the Boolean result. If it evaluates to true, then the event passes through the filter. Otherwise, the event does not pass into the filter window.

There are two ways to specify the Boolean filter associated with a filter window:

- through a C function that returns a **dfESPdatavar** of type int32 (return value != 0 ==> true; == 0 ==> false)
- by specifying an expression as a character string so that when it is evaluated it returns true or false

Examples

Example 1

The following example writes and registers a filter user-defined function:

```
// When quantity is >= 1000, let the event pass
//
//
dfESPdatavarPtr booleanScalarFunction(dfESPschema *is,
dfESPeventPtr ep, dfESPeventPtr oep) {

     // Get the input argument out of the record.
     dfESPdatavar dv(dfESPdatavar::ESP_INT32);
     // Declare a dfESPdatavar that is an int32.
     ep->copyByIntID(2, dv); // extract field #2 into the datavar

     // Create a new dfESP datavar of int32 type to hold the
     //      0 or 1 that this Boolean function returns.
     //
     dfESPdatavarPtr prv = new dfESPdatavar(dfESPdatavar::ESP_INT32);

     // If field is null, filter always fails.
     //
     if (dv.isNull()) {
          prv->setI32(0); // the return value to 0
     } else {
          // Get the int32 value from the datavar and compare to 1000
          if (dv.getI32() < 1000) {
               prv->setI32(0);     // set return value to 0
```

```
        } else {
            prv->setI32(1);     // set return value to 1
        }
    }
    return prv; // return it.
```

Place the following code inside **`main()`**:

```
dfESPwindow_filter *fw_01;
    fw_01 = cq->newWindow_filter("filterWindow_01", edm,
                                 dfESPindextypes::pi_RBTREE);
    fw_01->setFilter(booleanScalarFunction);
      // Register the filter UDF.
```

The **`setFilter`** function calls the filter function named booleanScalarFunction that you had previously registered.

Example 2

The following code example uses filter expressions.

```
    dfESPwindow_filter *fw_01;
    fw_01 = cq->newWindow_filter("filterWindow_01", edm,
                                 dfESPindextypes::pi_RBTREE);
    fw_01->setFilter("quant>=1000");
    // Register the filter expression.
```

For more information about user-supplied filter expressions, see the *DataFlux Expression Language: Reference Guide*.

dfESPwindow_copy

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.

Syntax

dfESPwindow_copy **windowID*;
windowID=contquery_name—> newWindow_copy
("*window_label*", *depot_name*,
dfESPIndextypes::*index*,*schema*);

Required Arguments

windowID
user-supplied identifier of the window to be copied

contquery_name
user-supplied name of the continuous query object specified in **`dfESPcontquery`**

window_label
user-supplied description of the window

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

schema
user-supplied name of the schema as specified by **`dfESPstring`**.

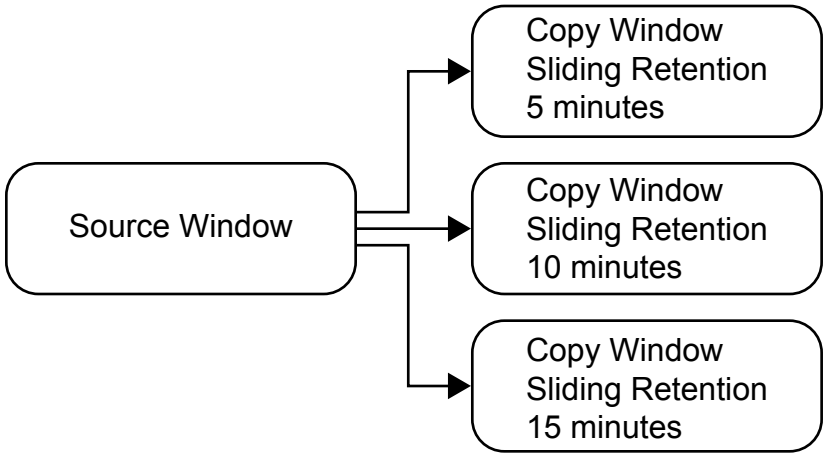
Details

You can set event state retention for copy windows 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.

Set the retention type on a window with the **`setRetentionParms()`** call. You can set type by count or time, and as either jumping or sliding. For more information, see "Retention Policies for Fully Stateful Indexes" on page 214.

The following figure depicts the application of a retention type on three copy windows that branch off the same source window. The time interval varies across the copy windows, but they all use sliding retention.

Figure 3.1 *Application of Sliding Retention with Varying Time Intervals to Copy Windows*



Example

Here is an example of how to specify a copy window:

```
dfESPwindow_copy *cw;
cw = cq->newWindow_copy("myCopyWindow", edm,
     dfESPindextypes::pi_HASH, sch);
```

Before creating this copy window, you use **`dfESPstring`** to specify a schema. For example

```
dfESPstring sch = dfESPstring("ID*:int32,symbol:string,price:double");
```

You can set event state retention for copy windows 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.

Set the retention type on a window with the **`setRetentionParms()`** call. You can set type by count or time, and as either jumping or sliding. For more information, see "Retention Policies for Fully Stateful Indexes" on page 214.

dfESPwindow_compute

defines a compute window, which enables a one-to-one transformation of input events to output events though the computational manipulation of the input event stream fields.

Syntax

dfESPwindow_compute **windowID*;
windowID=*contquery_name*—> newWindow_compute("*window_label*", *depot_name*, dfESPIndextypes::*index*, *schema*);

Required Arguments

windowID
user-supplied identifier of the compute window

contquery_name
user-supplied name of the continuous query object specified in **`dfESPcontquery`**

window_label
user-supplied description of the window

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

schema
user-supplied name of the schema as specified by **`dfESPstring`**.

Details

The compute window can be used to project input fields from one event to a new event and to augment the new event with fields that result from a calculation. The set of key fields can be changed within the compute window, but this capability must be used with caution. When a key field change is made within the compute window, the new set of keys must be opcode-compatible with the set of keys from the input streams. That is, Inserts, Updates and Deletes with respect to the input events' keys must be equivalent Inserts, Updates and Deletes with respect to the new key set.

Though the keys of a compute window are obtained from the keys of its input window, key values can be changed by designating the new fields as key fields in the **`dfESPcompute_window`** schema. When you change the key value in the compute window, the new key must also form a primary key for the window. If it does not, you might encounter errors because of unsuccessful Insert, Update, and Delete operations.

Examples

Example 1

Here is an example of a specification of a compute window:

```
dfESPwindow_source *cw;
cw = cq->newWindow_compute("myComputeWindow", edm,
     dfESPindextypes::pi_HASH, sch);
```

As with the source window, you use **dfESPstring** to specify a schema. For example

```
dfESPstring sch = dfESPstring("ID*:int32,symbol:string,price:double");
```

A compute window needs a field calculation method registered for each non-key field so that it computes the field value based on incoming event field values. These field calculation methods can be specified as either of the following:

- a collection of function pointers to C or C++ functions that return **dfESPdatavar** values and are passed an event as input
- expressions that use the field names of the input event to compute the values for the derived event fields

Example 2

The following example creates a compute window using a collection of function pointers.

Assume the following schema for input events:

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

The compute window passes through the input symbol and price fields. Then it adds a computed field (called cost) to the end of the event, which multiplies the price with the quantity.

A scalar function provides the input event and computes *price* * *quantity*. Functions that take events as input and returns a scalar value as a **dfESPdatavar** use a prototype of type **dfESPscalar_func** that is defined in the header file **dfESPfuncptr.h**.

Here is the scalar function:

```
dfESPdatavar *priceBYquant(dfESPschema*is, dfESPevent *nep,
      dfESPevent *oep, dfESPscontext *ctx) {
   //
   // If you are getting an update, then nep is the updated
   //      record, and oep is the old record.
   //
   // Create a null return value that is of type double.
   //
   dfESPdatavar *ret = new dfESPdatavar(dfESPdatavar::ESP_DOUBLE);
   // If you were called because a delete is being issued, you do not
   //      compute anything new.
   //
   if (nep->getOpcode() == dfESPeventcodes::eo_DELETE)
      return ret;
   void *qPtr = nep->getPtrByIntIndex(2); // internal index of
        quant is 2
   void *pPtr = nep->getPtrByIntIndex(3); // internal index of
        price is 3
   if ((qPtr != NULL) && (pPtr != NULL)) {
      double price;
```

```
        memcpy((void *) &price, pPtr, sizeof(double));
        int32_t quant;
        memcpy((void *) &quant, qPtr, sizeof(int32_t));
        ret->setDouble(quant*price);
    }
    return ret;
}
```

Note the **dfESPscontext** parameter. This parameter contains the input window pointer, the output schema pointer, and the ID of the field in the output schema computed by the function. Parameter values are filled in by the event stream processing engine and passed to all compute functions. Go to **$DFESP_HOME/src/compute_context** for another example that shows how to use this parameter.

The following code defines the compute window and registers the non-key scalar functions:

```
dfESPstring sch = dfESPstring
     ("ID*:int32,symbol:string, price:double,cost:double");

dfESPwindow_compute *cw;
cw = cq->newWindow_compute("myComputeWindow", edm,
                           dfESPindextypes::pi_HASH, sch);

// Register as many function pointers as there are non-key
//      fields in the output schema. A null for non-key
//      field j means copy non-key field j from the input
//      event to non-key field j of the output event.
//
cw->addNonKeyFieldCalc((dfESPscalar_func)NULL); // pass
     through the symbol
cw->addNonKeyFieldCalc((dfESPscalar_func)NULL); // pass
     through the price value
cw->addNonKeyFieldCalc(priceBYquant); // compute
     cost = price * quantity
```

This leaves a fully formed compute window that uses field expression calculation functions.

Example 3

The following example creates a compute window using field calculation expressions rather than a function. It uses the same input schema and compute window schema with the following exceptions:

1. You do not need to write field expression calculation functions.
2. You need to call **addNonKeyFieldCalc()** using expressions.

Note: Defining the field calculation expressions is typically easier. Field expressions can perform slower than calculation functions.

```
dfESPstring sch = dfESPstring
     ("ID*:int32,symbol:string,price:double,cost:double");

dfESPwindow_compute *cw;
cw = cq->newWindow_compute("myComputeWindow", edm,
                           dfESPindextypes::pi_HASH, sch);

// Register as many field expressions as there are non-key
```

```
//      fields in the output schema.
cw->addNonKeyFieldCalc("symbol"); // pass through the symbol
     value
cw->addNonKeyFieldCalc("price"); // pass through the price
     value
cw->addNonKeyFieldCalc("price*quantity"); // compute cost
     = price * quantity
```

Note: The field calculation expressions can contain references to field names from the input event schema. They do not contain references to fields in the compute window schema. Thus, you can use similarly named fields across these schemas (for example, symbol and price).

Note: Currently, you cannot specify both field calculation expressions and field calculation functions within a given window.

For more information, see the *DataFlux Expression Language: Reference Guide*.

dfESPwindow_union

specifies a simple join that merges one or more streams with the same schema.

Syntax

dfESPwindow_union **ID*
ID=*contquery_name*—> newWindow_union("*window_label*",
depot_name, dfESPIndextypes::*index*, true | false);

Required Arguments

ID
user-supplied identifier of the join

contquery_name
user-supplied name of the continuous query object specified in **`dfESPcontquery`**

window_label
user-supplied description of the union window

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

true | false
the strict flag — true for strict union and false for loose unions

Example

Here is an example of how to create a union window:

```
dfESPwindow_union *uw;
uw = cq->newWindow_union("myUnionWindow", edm,
     dfESPindextypes::pi_HASH, true);
```

All input windows to a union window must use 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.

dfESPwindow_aggregate

specifies an aggregation window. An aggregation window is similar to a compute window in that non-key fields are computed.

Syntax

dfESPwindow_aggregate **windowID*;
windowID=*contquery_name*—> newWindow_aggregate("*window_label*", *depot_name*, dfESPIndextypes::*index*, *schema*);

Required Arguments

windowID
user-supplied identifier of the aggregate window

contquery_name
user-supplied name of the continuous query object specified in **dfESPcontquery**

window_label
user-supplied description of the window

depot_name
user-supplied name of the depot as specified by **dfESPeventdepot**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

schema
user-supplied name of the aggregate schema. Specify an aggregate schema the same as you would for any other window schema, except that key field(s) are the group-by mechanism.

See Also

Chapter 5, "Creating Aggregate Windows," on page 73

dfESPwindow_join

specifies a join window, which takes two input windows and a join type.

Syntax

dfESPwindow_join **windowID*;
windowID=contquery_name—> newWindow_join("*window_label*",
dfESPwindow_join::*jointype*, *depot_name*, dfESPIndextypes::*index*);

Required Arguments

windowID
user-supplied identifier of the join window

contquery_name
user-supplied name of the continuous query object specified in **`dfESPcontquery`**

window_label
user-supplied description of the window

jointype
type of join to be applied

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

See Also

"Creating Join Windows" on page 77

dfESPwindow_pattern

enables the detection of events of interest. A pattern defined in this window type is an expression that logically connects declared events of interest.

Syntax

dfESPwindow_pattern **windowpattern*;
windowpattern=contquery_name—> newWindow_pattern("*label*"
depot_name, dfESPIndextypes::*index*, dfESPstring(*schema*)), ;

Required Arguments

windowpattern
user-supplied name of the window pattern to detect

contquery_name
user-supplied name of the continuous query object specified in **`dfESPcontquery`**

label
user-supplied description of the window pattern

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

schema
schema associated with the window feeding the pattern window

See Also

Chapter 7, "Creating Pattern Windows," on page 83

dfESPwindow_procedural

enables the specification of an arbitrary number of input windows and input handler functions for each input window.

Syntax

dfESPwindow_procedural **windowID*;
name= *query_name*—> newWindow_procedural
("*label*", *depot_name*,
dfESPIndextypes::*index*, *schema*);

Required Arguments

windowID
user-supplied identifier of the procedural window

query_name
user-supplied name of the query object specified in **`dfESPcontquery`**

label
user-supplied description

depot_name
user-supplied name of the depot as specified by **`dfESPeventdepot`**

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

schema
schema defined by **`dfESPstring`**

See Also

Chapter 8, "Creating Procedural Windows," on page 97

dfESPwindow_textContext

enables the abstraction of classified terms from an unstructured string filed. Results in event for each classified term.

Syntax

dfESPwindow_textContext **windowID*;
windowID=query_name—> newWindow_textContext("*window_label*", *depot_name*, dfESPIndextypes::*index*, *litiFiles*, *inputFieldName*);

Required Arguments

windowID
user-supplied ID of the text context window

query_name
user-supplied name of the query object that is specified with **`dfESPcontquery`**

window_label
user-supplied description of the window

index
primary index. Six types of primary index are supported. For more information, see "Understanding Primary Indexes and Retention Policies" on page 213.

litiFiles
comma separated list of Liti file paths.

inputFieldName
user-supplied name for the string filed in the input event to analyze

Details

This object enables users who have licensed SAS Text Analytics to use its Liti files to initialize a **`textContext`** 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 **`textContext`** window to look for tweet patterns of interest.

Example

The following example uses an empty index so that the **`textContext`** window, which is insert only, does not grow endlessly. It follows the **`textContext`** window with a copy window that uses retention to control the growth of the classified terms.

```
// Build the source window. We specify the window name, the schema
        //    for events, the depot used to generate the index and handle
        //    event storage, and the type of primary index, in this case a
        //    hash tree
        //
        dfESPwindow_source *sw_01;
        sw_01 = cq_01->newWindow_source("sourceWindow_01", depot_01,
                        dfESPindextypes::pi_HASH,
                        dfESPstring("ID*:int32,tstamp:date,msg:string"));

        // Build the textContext window. We specify the window name, the depot
        //    used for retained events, the type of primary index, the Liti
        //    files specification string, and the input string field to analyze.
        // Note that the index for this window is an empty index.  This means
        //    that events created in this window will not be retained in this
        //    window.  This is because textContext windows are insert only,
        //    hence there is no way of deleting these events using retention
        //    so without using an empty index this window would grow indefinitely.
```

```
// If you try to run this example you will need to have licensed SAS
//    Text Analytics, whose install contains these Liti language files.
// You will need to change the litFiles string below to point to your
//    installation of the Liti files otherwise the text analytics engine
//    will not be initialized and classified terms will not be found.
//
dfESPwindow_textContext *tcw_01;

dfESPstring litiFiles = "/wire/develop/TableServer/src/common/dev/
                         mva-v940m1/tktg/misc/en-ne.li,
                         /wire/develop/TableServer/src/common/dev/
                         mva-v940m1/tktg/misc/ro-ne.li";

// We are placing a copy window after the textContext window so that
//    it can be used to hold the textContext events with an established
//    retention policy. This is a design pattern for insert-only windows.
tcw_01 = cq_01->newWindow_textContext("textContextWindow_01", depot_01,
                           dfESPindextypes::pi_EMPTY, litiFiles, "msg");

// Create the copy window.
dfESPwindow_copy *cw_01;
cw_01 = cq_01->newWindow_copy("copyWindow_01", depot_01,
                                dfESPindextypes::pi_RBTREE);

// Now set its retention policy to a sliding window of 5 mins.
// This example only has 3 events being injected so the retention
//    policy will not take effect, but if we published enough data
//    into this model then it would start retaining older events out
//    using retention deletes once they aged past 5 mins.
cw_01->setRetentionParms(dfESPindextypes::ret_BYTIME_SLIDING, 300);
```

Suppose you supply the following strings to the **`textContext`** window:

```
"i,n,1,2010-09-07 16:09:01,I love my Nissan pickup truck"
"i,n,2,2010-09-07 16:09:21,Jerry went to dinner with Renee for last Sunday"
"i,n,3,2010-09-07 16:09:43,Jennifer recently got back from Japan where
 she did game design project work at a university there"
```

Here are the results.

```
event[0]: <I,N: 1,0,Nissan pickup,13,VEHICLE,7,10,22,3,5>
event[1]: <I,N: 2,0,Sunday,6,DATE,4,41,46,8,9>
event[2]: <I,N: 2,1,Renee,5,PROP_MISC,9,26,30,5,6>
event[3]: <I,N: 3,0,Japan,5,LOCATION,8,32,36,5,6>
event[4]: <I,N: 3,1,game design project work,24,NOUN_GROUP,10,52,75,9,13>
event[5]: <I,N: 3,2,Jennifer,8,PROP_MISC,9,0,7,0,1>
```

dfESPdatavar

represents a variable of any of the SAS Event Stream Processing Engine data types

Syntax

dfESPdatavar **name* = new dfESPdatavar(dfESPdatavar::*data_type*);

Required Arguments

name
user-supplied name of the variable

data_type
Can be one of the following values:

- **ESP_INT32**
- **ESP_INT64**
- **ESP_DOUBLE (IEEE)**
- **ESP_UTF8STR**
- **ESP_DATETIME** (second granularity)
- **ESP_TIMESTAMP** (microsecond granularity)
- **ESP_MONEY** (192-bit fixed decimal)

A **dfESPdatavar** of any of these types can be NULL. Two **dfESPdatavars** that are each NULL are not considered equal if the respective types do not match.

Example

Create an empty **dfESPdatavar** and then set the value as follows:

```
dfESPdatavar *dv = new dfESPdatavar(dfESPdatavar::ESP_INT32);
dv->setI32(13);
```

Get access to raw data in the **dfESPdatavar** using code like this:

```
void *p = dv->getRawdata(dfESPdatavar::ESP_INT32);
```

This returns a pointer to actual data, so in this **int32** example, you can follow with code like this:

```
int32_t x;
memcpy((void *)&x, p, sizeof(int32_t));
```

This copies the data out of the **dfESPdatavar**. You can also use the **getI32** member function (a get and set function exists for each data type) as follows:

```
int32_t x;
x = dv->getI32();
```

Many convenience functions are available and a complete list is available in the modeling API documentation included with the installation.

dfESPschema

represent the name and types of fields, as well as the key structure of event data

Syntax

dfESPschema **name*= new dfESPschema(*schema*);

Required Arguments

name
user-supplied name of the representation

schema
specifies the structure of fields of event data

Details

A **dfESPschema** never represents field data, only the structure of the fields. When a **dfESPschema** object is created, it maintains the field names, fields types, and field key designation in the original order the fields (called the external order) and in the packing order (called the internal order) for the fields.

SAS Event Stream Processing Engine does not put restrictions on the field names in a schema. Even so, you need to keep in mind that many times field names are used externally. For example, there could be connectors or adapters that read the schema and use field names to reference external columns in databases or in other data sources. It is therefore highly recommenced that you start field names with an alphanumeric character, and use no special characters within the name.

In external order, you specify the keys to be on any fields in the schema. In internal order, the key fields are shifted left to be packed at the start of the schema. For example, using a compact character representation where an "*" marks key fields, you specify this:

```
"ID1*:int32,symbol:string,ID2*:int64,price:double"
```

This represents a valid schema in external order. If you use this to create a **dfESPschema** object, then the object also maintains the following:

```
"ID1*:int32, ID2*:int64,symbol:string,price:double"
```

This is the same schema in internal order. It also maintains the permutation vectors required to transform the external form to the internal form and vice versa.

Creating a **dfESPschema** object is usually completed by passing the character representation of the schema to the constructor in external order, for example:

```
dfESPschema *s = new
     dfESPschema("mySchema","ID1*:int32,symbol:string,ID2*:
     int64,price:double");
```

A variety of methods are available to get the names, types, and key information of the fields in either external or internal order. There are also methods to serialize the schema back to the compact string form from its internal representation.

dfESPevent

creates a packed binary representation of a set of field values.

Syntax

dfESPevent **name*= new dfESPevent(*schemaPtr*, *charEvent*);

Required Arguments

name
user-supplied name of the event

schemaPtr
user-supplied schema pointer

charEvent
{i|u|p|d},{n|s},f1,f2,...,fn where

i | u | p | d	means Insert, Update, Upsert, and Delete respectively
n | p	means normal event or partial-update event
f1, ..., fn	are the fields that make up the data portion of the event

Details

Data in an **dfESPevent** object is stored in internal format (as described in the **dfESPschema** object), so all key values are contiguous and packed at the front of the event. The **dfESPevent** 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 **dfESPevents** created using the same underlying schema.

Both metadata and field data are associated with an event. The metadata consists of the following:

- an opcode (indicating whether the event represents an Insert, Update, Delete, or Upsert)
- 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 microsecond **timestamps** that can be used for latency measurements

The **dfESPevent** class has member functions for both accessing and setting the metadata associated with the event. For information about these functions, see the detailed class and method documentation that is available at **$DFESP_HOME/doc/html**.

The field data portion of an event is also accessible from the **dfESPevent** in the following ways:

- Event field data can be copied out of an event into a **dfESPdatavar** using the **copyByIntID()** or **copyByExtID()** methods.
- A **dfESPdatavar** can be set to point into the allocated memory of the **dfESPevent** using the **getByIntID()** or **getByExtID()** methods.
- A pointer into the **dfESPevent** packed field data can be obtained through the **getPtrByIntIndex()** method.

To assure the best performance, work with binary events whenever possible.

Additional aspects of the **dfESPevent** class include the ability to do the following:

- Write a compact serialized form of the event to a file using the **fwrite()** method.
- Read in the serialized event into a memory buffer through the **getSerializeEvent()** method.
- Create a new event by passing the serialized version of the event to the **dfESPevent** constructor.

See also "Converting CSV Events to Binary" on page 208.

dfESPeventblock

creates a lightweight wrapper around a collection of events

Syntax

dfESPeventblock :: newEventBlock
(&*name*,
dfESPeventblock:: ebtTRANS | ebtNORMAL);

Required Argument

name
user-supplied pointer to a contained **`dfESPevent`**

Details

Generate a **`dfESPeventblock`** object by publishing clients. An event block is maintained as it is passed between windows in an application, as well as to subscribing clients. The **`dfESPeventblock`** object can report the number of items that it contains and return a pointer to a contained **`dfESPevent`** when given an index.

A unique embedded transaction ID is generated for event blocks as they are absorbed into a continuous query. Event blocks can also be assigned a unique ID by the publisher. In addition to the event block ID, the publisher can set a host and port field in event blocks to establish where the event block is coming from. This meta information is used by the guaranteed delivery feature to ensure that event blocks make their way from a publisher.

Event blocks progress through the continuous queries and on to one or more guaranteed subscribers. The event block meta information is carried with the event block from the start of processing at a source window. The meta information progresses through all stages of computation in derived windows and on to any subscribing clients. You can use the publisher assigned ID, host, and port to tie an output **`dfESPeventblock`** back to an input **`dfESPeventblock`**.

Create new **`dfESPeventblock`** objects with either transactional (**`dfESPeventblock::ebt_TRANS`**) or normal (**`dfESPeventblock::ebt_NORMAL`**) event semantics. Transaction semantics imply that each **`dfESPevent`** contained in the block must be able to be applied to the index in a given window. Otherwise, none of the events are applied to the index.

For example, suppose an **`dfESPeventblock`** has 100 events and the first event is a delete event. Further suppose that the delete event fails to be applied because the underlying event to be deleted is not present. In that case, the remaining 99 events are ignored, logged, and written to a bad records file (optional). Normal semantics imply that each event in a **`dfESPeventblock`** is treated as an individual event. Therefore, the failure for one event to apply to an index causes only that event to not be incorporated into the window.

A **`dfESPeventblock`** with more than one event, but without transactional properties set, can be used to improve performance during the absorption of the event block into the appropriate source window. You use this to trade off a little bit of latency for a large gain in throughput. It is best to test the event block optimal size trade-off. For example,

placing 256 events into an event block gives both great latency and throughput. This performance varies depending on the width of the events.

dfESPpersist

persists and restores an engine instance that has been shutdown. The instance is restored to the exact state that it was in when it was persisted.

Syntax

dfESPpersist *object*(“*directory*”);

Required Arguments

object
user-supplied name of the object to be persisted and later restored.

directory
user-supplied name of the directory where the object is stored

Examples

Example 1

The following code persists and engine. The directory specified to the **`dfESPpersist`** object is created if it does not exist. Any data injected into the **`dfESPengine`** after the **`dfESPpersist::persist()`** call is not part of the saved engine.

```
// Assume that the engine is running and injecting data, ....

    // Declare a persist/restore object
    //    "PERSIST" is the directory name where the persisted copy
    //    is stored. This can be a relative or absolute path
    //
    { // persist object must reside in a scoped block because of a known
      // destructor issue
    bool success = dfESPpersist persist_restore("PERSIST");
    if (!success) {
       //
       //    Handle any error -- the engine might not be persisted fully or
       //    might be only partially persisted. The running engine is fine
       //    and not compromised, but the persisted snapshot is not valid.
       //
    }
    }  // end of scoped block
    // Tell the persist/restore object to create a persisted copy.
    //
    persist_restore.persist();
```

For more information, see “Persist and Restore Operations” on page 222.

Example 2

Restore state from a persisted copy of an engine by starting your event stream processing application with the following command line option:**`–r persisted_path`** where *persisted_path* can be a relative or absolute path.

When your application makes a **`dfESPengine::startProjects()`** call, the engine is restored from the persisted state, and all projects are started.

```
// Construct the model but do not start it.
          ....
     // Declare a persist/restore object
     //    "PERSIST" is the directory name where the persisted copy
     //    had been saved. This can be a relative or absolute path
     //
     dfESPpersist persist_restore("PERSIST");
     // Tell the persist/restore object to reload the persisted engine
     //
     bool sucess = persist_restore.restore();
     if (!success) {
        //
        //    Handle any error -- the engine might not be restored fully
        //    or it could be partially restored, or in a compromised state.
        //
     }
     // Start all projects from restored state of engine.
     dfESPengine::startProjects();
```

Chapter 4

Using the XML Modeling Layer

Overview to the XML Modeling Layer

The XML modeling layer for SAS Event Stream Processing Engine is a higher-level abstraction of the C++ Modeling API. The XML modeling layer enables someone without a background in programming to build event stream processor models.

The high-level syntax of XML models is as follows:

```
<engine>
    <projects>
        +<project>
            <contqueries>
                +<contquery>
                    <windows>
                        +<window-type> </window-type>
                    </windows>
                    <edges>
                        +<edge> </edge>
                    </edges>
                </contquery>
            </contqueries>
        </project>
    </projects>
</engine>
```

The **<window-*type*>** element can be one of the following:

- **<window-source>**
- **<window-copy>**
- **<window-filter>**
- **<window-compute>**
- **<window-aggregate>**
- **<window-join>**
- **<window-union>**
- **<window-procedural>**
- **<window-pattern>**

These window elements can contain other elements. For more information, see "Dictionary of XML Elements" on page 44.

Using the XML Modeling Server

The XML modeling server is an executable that instantiates and executes an engine model that contains zero or more projects. It supports server control communication through a socket interface. The socket interface uses a server port number that is defined when the server starts through the engine model or the command line option.

Users or applications read and write XML using this socket to interact with a running event stream processing engine on the XML factory server. You can use this interface to start, stop, create, and remove projects. You can also use it to publish events and query windows.

Use the **$DFESP_HOME/bin/dfesp_xml_server** command to start an XML server.

Option	Description
-properties *file*	Specify a properties file, which can contain the following entries: • **esp.pubsub.port**=*n* Specifies the publish/subscribe port, and has the same effect as specifying the -port *n* option to the command. • **esp.server.port**=*n* Specifies the XML server port, and has the same effect as specifying the -server *n* option to the command. • **esp.dateformat**=*f* Specifies the **strptime** string, and has the same effect as specifying the -dateformat *f* option to the command.
-port *n* | **-pubsub *n***	Specifies the publish/subscribe port.
-server *n*	Specifies the XML server port.
-dateformat *f*	Specifies the **strptime**. string

Option	Description
-messages ***dir***	Specifies the directory in which to search for localized message files.
-xsd ***file***	Specifies the schema file for validation.
-model ***url***	Specifies the url to XML model.
-plugindir ***dir***	Specifies the directory in which to search for dynamically loaded plug-ins, defaults to **plugins**.
-logconfig ***file***	Specifies the SAS logging facility configuration file.
-loglevel off | trace | debug | info | warn | error | fatal	Sets the logging level.

Using the XML Modeling Client

Starting the Client

Use the following command to start an XML modeling client: **$DFESP_HOME/bin/dfesp_xml_client**.

The client enables you to send XML fragments to a running XML server process that was started with the **dfesp_xml_server** command. XML fragments enable the injection and removal of events and the starting and stopping of projects. You can inject events into a running project or submit a project data set. This data set can inject a project, run commands, return results, and then remove the project.

Option	Description
-server ***host*:*port***	Specify the *host*and *port* of a currently running XML server.
-file ***input_file***	Specify an XML file that contains code to load, start, stop, or remove projects.

Managing Projects with XML Code

You can load, start, stop, and remove projects through XML code.

Action	XML code

Load a project	`<project action='load' name='events' pubsub='auto'>`
	`  <!-- any valid project contents -->`
	`</project>`

Start a project	`<project name='reference' action='start' />`
Stop a project	`<project name='reference' action='stop' />`
Remove a project	`<project name='reference' action='remove' />`
Persist a project	`<persist project='reference' path='put_persisted_project'/>`
Persist all projects	`<persist path='put_all_persisted_projects'/>`

Restore a project	`<project action='load' name='events'`
	`  restore='get_persisted_project'>`
	`  <!-- any valid project contents -->`
	`</project>`

Note: The body of the project must be the same that was used when persisting the project.

You can inject events into a running event stream processing engine. The XML server supports Insert, Update, Upsert, and Delete opcodes.

To inject events into a project, use code such as this:

```
<prime>
        <inject name='event-input' project='reference'
                contquery='events' window='events'/>
    </prime>
    <event-streams>
      <xml-stream name='event-input'>
           <events>
               <event opcode='insert'>
                   <value name='id'>10</value>
                   <value name='user'>bob</value>
                   <value name='date'>1/aug/2013:08:00:00</value>
               </event>
               <event opcode='insert'>
                   <value name='id'>20</value>
                   <value name='user'>scott</value>
                   <value name='date'>1/aug/2013:09:00:00</value>
               </event>
           </events>
      </xml-stream>
     </event-streams>
```

For example, combined model and event processing enables you to send a model or project and then inject data as specified in event-stream element into the model. Then the XML server responds with the result. The project lives for only the request. It is removed after the request. The model configuration file is not required in this case.

The following code shows combined model and event processing. It sends a project over an event stream and captures window contents after the project run.

```
<stream>
            <contqueries>
                  <!-- any vaild contequeries contents -->
            </contqueries>

            <prime>
              <inject stream='events-input' contquery='events' window='events'/>
            </prime>
            <event-streams>
              <xml-stream name='events-input'>
                <events>
                    <event opcode='insert'>
                        <value name='id'>10</value>
                        <value name='user'>larry</value>
                    </event>
                    <event opcode='insert'>
                        <value name='id'>20</value>
                        <value name='user'>moe</value>
                    </event>
                    <event opcode='insert'>
                        <value name='id'>30</value>
                        <value name='user'>curly</value>
                    </event>
                    <event opcode='insert'>
                        <value name='id'>40</value>
                        <value name='user'>curly</value>
                    </event>
                    <event opcode='insert'>
                        <value name='id'>50</value>
                        <value name='user'>moe</value>
                    </event>
                    <event opcode='insert'>
                        <value name='id'>60</value>
                        <value name='user'>moe</value>
                    </event>
                </events>
              </xml-stream>
            </event-streams>
          </stream>
```

Here is what is output from the previous stream:

```
<response elapsed='1006 ms'>
                <events name='users'>
                        <columns>
                            <column label='user' name='user'
                             sqltype='12' type='varchar'/>
                            <column label='numEvents' name='numEvents'
                             numeric='true' sqltype='4' type='integer'/>
                        </columns>
```

```
                <results>
                    <data>
                        <value column='user'>moe</value>
                        <value column='numEvents'>3</value>
                    </data>
                    <data>
                        <value column='user'>curly</value>
                        <value column='numEvents'>2</value>
                    </data>
                    <data>
                        <value column='user'>larry</value>
                        <value column='numEvents'>1</value>
                    </data>
                </results>
            </events>
        </response
```

Using the XML Modeling Validation Tool

Use the following command to start the XML modeling validation tool:
`$DFESP_HOME/bin/dfesp_xml_validate "`*`file_to_validate`*`"`

The validation script uses the Model.rnc XML schema definition file and the Jing.jar validation code to perform a syntactic check on the specified XML model file.

When the validation tool finds any part of the specified model to be in violation of the schema definition, it generates error messages, including line numbers and descriptions.

Dictionary of XML Elements

Here is an alphabetically ordered list of the XML elements that you can use for the XML modeling layer. For each element listed, the following information is provided:

- the required and optional elements that the element can contain
- the required and optional attributes of the element
- a usage example

Table 4.1 *XML Elements*

Element	Description	Details	
`conditions`	A list of left/ right field match pairs for joins.	Elements:	**`fields`**
		Attributes:	Enclosed text specifies a list of one or more field elements that specify the equijoin conditions.
		Example:	See "Window-Join Examples" on page 70.

Element	Description	Details	
connector	A publish or subscribe source-sink for events.	Elements:	**properties** Each **property name=*value*** specified within a **properties** element encloses text that specify a valid connector property value.
		Attributes:	**class = fs** | **db** | **mq** | **pi** | **smtp** | **sol** | **tdata** | **tibrv** | **tva** For more information about these connector classes and valid connector properties, see "Using Connectors" on page 129. **type= publish** | **subscribe**

		Example:	`<connectors>`
			`  <connector class='fs' type='publish>`
			`  <properties>`
			`  <property name='fstype'>`
			`  syslog</property>`
			`  <property name='fsname'>`
			`  data/clean.log.bi</property>`
			`  <property name='growinginputfile'>`
			`  true</property>`
			`  <property name='transactional'>`
			`  true</property>`
			`  <property name='blocksize'>`
			`  128</property>`
			`  </properties>`
			`  </connector>`
			`</connectors>`

connectors	A list of connector elements.	Elements:	One or more **connector** elements.
		Attributes:	None.
		Example:	See **connector**.
context-plugin	A wrapper for a shared library and function name. The function, when called, returns a **dfESPpcontext** for procedural windows and a **dfESPscontext** for compute windows.	Elements:	None.
		Attributes:	**name=*sharedlib*** The shared library that contains the context generation function. **function=*name*** The function that when called returns a new derived context for the procedural window's handler routines.
		Example:	See "XML Examples of Procedural Windows" on page 103.

Element	Description	Details	
contqueries	A container for the list of **contquery** elements.	Elements:	One or more **contquery** elements.
		Attributes:	None.

		Example:	`<contqueries>`
			`  <contquery> ... </contquery>`
			`  <contquery> ... </contquery>`
			`</contqueries>`

contquery	The definition of a continuous query, which includes windows and edges.	Elements	**windows** [**edges**]
		Attributes:	**name=*string*** The name of the continuous query. [**trace=*string***] One or more space or comma-separated window names or IDs. [**index= pi_RBTREE | pi_HASH | ph_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**] A default index type for all windows in the continuous query that do not explicitly specify an index type. **[timing-threshold=*value*]** When a window in the query takes more than *value* microseconds to compute for a given event or event block, log a warning message.

		Example:	`<contquery name='cq1'>`
			`  <windows>`
			`  <window-type name='one'>...</window-type>`
			`  <window-type name='two'>...</window-type>`
			`  </windows>`
			`</contquery>`

ds2-code	A block of DS2 code that is used as an input handler for procedural windows.	Elements:	CDATA that provides a block of DS2 source code to serve as the handler function.
		Attributes:	**source=*file*** Value of the plug-in element.
		Example:	See "XML Examples of Procedural Windows" on page 103.

Element	Description	Details	
edge	The connectivity specification between two or more windows.	Attributes:	**source=*ID*** String to specify the window ID of the leading edge. **target=*ID*** String to specify one or more IDs of trailing edges, separated by spaces. [**slot=*int***] Integer to specify the slot to use with a splitter.

		Example:	`<edges>`
			`  <edge source='wind001' target'win002'/>`
			`  <edge source='wind002' target'win003'/>`
			`  <edge source='wind003' target'win004`
			`  win005 win006'/>`
			`  ...`
			`</edges>`

edges	A container for a list of **edge** elements.	Elements:	One of more **edge** elements.
		Attributes:	None.
		Example:	See **edge**.

Element	Description	Details	
engine	The global wrapper for an event stream processing engine.	Elements:	**projects** [**esp-server** [**port=***port*]] If the port is not specified here and a port is specified as an attribute of the engine element, use the publish/subscribe *port*+1 as the ESP XML server port.
		Attributes:	[**port=**] The publish/subscribe port for the engine [**dateformat**=] The date format to use when converting dates to or from an internal representation. [**on-project-fatal="exit"** | **"exit-with-core"** | **"stop-and-remove"]** Specify how the engine reacts to fatal errors in project. Do one of the following: • Exit with the engine process. • Exit and generate a core file for debugging, or stop all processing. • Disconnect publish/subscribe, clean up all threads and memory, and remove the process from the engine, leaving the engine up and processing other projects. .

		Example:	`<engine port='31417`
			`dateformat='YYYYMMDD HH:mm:ss'>`
			`<esp-server port='5000'/>`
			`<projects>`
			`...`
			`</projects>`
			`</engine>`

Element	Description	Details	
event	The definition of an event of interest (EOI) for pattern matching.	Elements:	None.
		Attributes:	**name=*ID*** or **source=*file*** User-specified ID for the event or data source. Enclosed text specifies a WHERE clause that is evaluated by the expression engine.
		Example:	See "XML Pattern Window Examples" on page 93.
events	A wrapper for a list of event elements.	Elements:	One or more **event** elements. **logic** **output** [**timefields**]
		Attributes:	None.
		Example:	See "XML Pattern Window Examples" on page 93.
expression	An interpreted expression in the DataFlux expression languages. Variables are usually fields from events or are variables declared in the **expr-initialize** element.	Elements:	None.
		Attributes:	Enclosed text is the expression to be processed.
		Example:	`<expression>quantity >= 100</expression>`
expr-initialize	An initialization expression code block, common to window types that allow the use of expressions.	Elements:	None.
		Attributes:	**type = 'int32' | 'int64' | 'double' | 'string' | 'money' |'date' | 'stamp'** Enclosed text specifies the block of code used to initialize the expression engine. Variables in the initialization block can be used in the filter expression.
		Example:	See "Window-Compute Examples" on page 69.

Element	Description	Details	
field	A definition or a reference to a column in an event.	Elements:	None.
		Attributes:	**type= int32** | **int64** | **double** | **string** | **money** | **date** | **stamp** **name=*field_name*** [**key= true** | **false**] The default is **false**.

		Example:	`<schema name='input_readings'>`
			` <fields>`
			`  <field type='int32' name='ID' key='true'/>`
			`  <field type='string' name='sensorName'/>`
			`  <field type='double' name='sensorValue'/>`
			` </fields>`
			`</schema>`

fields	A container for a list of field elements.	Elements:	One or more **field** elements.
			For **window-join**: none.
		Attributes:	None.
			For **window-join**: **left=*name***. A field name from the left input table. **right=*name*** A field name from the right input table.
		Example:	See **field**.
			For **window-join**, see "Window-Join Examples" on page 70.

Element	Description	Details	
field-expr	An algebraic expression whose value is assigned to a field.	Elements:	None.
		Attributes:	Enclosed text specifies the value of aggregate expression. Includes one or more of the following fields: • **ESP_aSum(*fieldName*)** • **ESP_aMax(*fieldName*)** • **ESP_aMin(*fieldName*)** • **ESP_aAve(*fieldName*)** • **ESP_aStd(*fieldName*)** • **ESP_aWAve(*fieldName*, *fieldName*)** • **ESP_aCount(*fieldName*)** • **ESP_aLast(*fieldName*)** • **ESP_aFirst(*fieldName*)** • **ESP_aLastNonDelete(*fieldName*)** • **ESP_aLastOpCode(*fieldName*)** • **ESP_aGUID()** • **ESP_aCountOpCodes(1|2|3)** For more information, see "Aggregate Functions for Aggregate Window Field Calculation Expressions" on page 216.
			For **window-join**, the enclosed text is any valid SAS DataFlux scalar expression that uses input field names and variables from the **expr-initialize** block. Prefix input field names from the left table with **l_** and input field names from the right input table with **r_**.
		Example:	See "Window-Compute Examples" on page 69.

Element	Description	Details	
field-plug	A function in a shared library whose returned value is assigned to a field.	Elements:	None.
		Attributes:	**plugin=*name*** Name of the shared library. **function=*name*** Name of the function in the shared library. [**additive = true** | **false**] Defaults to false.
		Example:	See "Window-Compute Examples" on page 69.
field-selection	A reference to a field whose value is assigned to another field.	Elements:	None.
		Attributes:	**name=*ID*** Selected field in the output schema. **source=*output_window_field*** The *output_window_field* takes the following form:**l_*field_name*** | ***rfield_name.***, where **l_** indicates that the field comes from the left window and **r_** indicates that the field comes from the right window.
		Example:	See "Window-Join Examples" on page 70.

Element	Description	Details	
join	A container in a join window that collects common join attributes and the **conditions** element.	Elements:	**conditions**
		Attributes:	**left=*name*** Name or ID of the left input table. **right=*name*** Name or ID of the right input table. **type = fullouter** | **leftouter** | **rightouter** | **inner** Join type. [**no-regenerates**] Do not regenerate join changes when the dimension table changes. **use-secondary-index = true** | **false** When true, automatically generate and maintain a secondary index to assist table computation when the dimension table changes.
		Example:	See "Window-Join Examples" on page 70.
logic	In a **pattern** element, a text string that represents the logical operator expression to combine events of interest (EOIs).	Elements:	None.
		Attributes:	Enclosed text specifies a prefix expression that defines the temporal pattern. For example: **fby(e1,fby(e2,not(e3)),e4,e5,e6)** when e1, e2, e3, e4, and e5 are named event elements.
		Example:	See "XML Pattern Window Examples" on page 93.
output	A wrapper that is used within several window subtypes to specify how window output is generated.	Elements:	**field-expr** | **field-plug** Specifically for **window-join**: **field-expr** | **field-plug** | **field-selection**
		Attributes:	None.
		Example:	See "Window-Compute Examples" on page 69.

Element	Description	Details	
pattern	A wrapper within a pattern window where events of interest (EOIs), controlling attributes, pattern logic expression, and pattern output rules are grouped.	Elements:	**events**
		Attributes:	[**index=*fields***] A comma-separated list of fields from the input windows that forms an index generation function.
		Example:	See "XML Pattern Window Examples" on page 93.
patterns	A container for a list of **pattern** elements.	Elements:	One or more **pattern** elements.
		Attributes:	None.
		Example:	See "XML Pattern Window Examples" on page 93.
plugin	A shared library and function name pair that specifies filter window functions and procedural window handlers.	Elements:	None.
		Attributes:	**name=*shared_lib*** Shared library that contains the specified function. **function=*name*** The specified function.
		Example:	See "XML Examples of Procedural Windows" on page 103.

Element	Description	Details	
project	The primary unit in the XML server. Contains execution and connectivity attributes and a list of continuous queries to execute.	Elements:	**contqueries**
		Attributes:	**name=***string* Project name. **threads=***int* A positive integer for the number of threads to use from the available thread pool. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the project. [**port=***port*] The project-level publish/subscribe port for auto or manual mode [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] A default index type for all windows in the project that do not explicitly specify an index type. [**useTaggedToken= true** | **false**] Specify whether tagged token data flow semantics should be used for continuous queries.

		Example:	`<project name='analysis' threads='16'`
			`pubsub='auto' port='31417'`
			`index='pi_HASH' useTaggedTokens='true'>`
			`<contqueries>`
			`...`
			`</contqueries>`
			`</project>`

projects	A container for a list of **project** elements.	Elements:	Zero or more **project** elements
		Attributes:	None

		Example:	`<projects>`
			`<project...> ... </project>`
			`<project...> ... </project>`
			`</projects>`

Element	Description	Details	
retention		Elements:	None.
		Attributes:	**type= bytime_jumping | bytime_sliding | bycount_jumping | bycount_sliding** Retention type. Enclosed text specifies the value of the retention element. When specifying a time-based retention policy, specify number of seconds. When specifying a count-based retention policy, specify a maximum row count. [**field=*name***] Name of a field of type datetime or timestamp. Used to drive the time-based retention policies. If not specified, wall clock time is used.
		Attributes:	**name=*string*** Name of shared library containing the splitter function. **function=*function_name***
		Example:	`<splitter-plug name='libmethod' function='splitter'/>`
		Example:	See "Window-Copy Example" on page 69.
schema	A named list of fields.	Elements:	**fields**
		Attributes:	[**name=*schema_name***]
		Example:	See **field**.
schema-string	Compact notation to specify a schema.	Elements:	None.
		Attributes:	Enclosed text specifies the schema.
		Example:	`<schema-string>ID**"int32, sensorName:string, sensorValue:double </schema-string>`

Element	Description	Details	
splitter-expr	A wrapper to define an expression that directs events to one of *n* different output slots.	Elements:	**expression** Value of the expression element. [**expr-initialize type=*return_type***] Initialization expression return type.

		Example:	`<splitter-expr>`
			`  <expr-initialize type='int32'>`
			`  <![CDATA[`
			`  integer counter`
			`  counter=0`
			`  ]]>`
			`  </expr-initialize>`
			`  <expression>counter%2</expression>`
			`</splitter-expr>`

splitter-plug	An alternative way to specify splitter functions using a shared library and a function call.	Attributes:	**name=*string*** Name of shared library containing the splitter function. **function=*function_name*** Name of the function call.

		Example:	`<splitter-plug`
			`  name='libmethod'`
			`  function='splitter'/>`

Element	Description	Details	
timefield	The field and source pair that specifies the field in a window to be used to drive the time (instead of clock time).	Elements:	None.
		Attributes:	**field=*name*** User-specified name of the time field. **source=*window*** Name of an input window to which to attach this time field.
		Example:	`<timefield field='firstrun' source='win1'/>`
timefields	A container of a list of **timefield** elements.	Elements:	**timefield**
		Attributes:	None.

		Examples:	`<timefields>`
			`  <timefield>...</timefield>`
			`</timefields>`

Element	Description	Details	
windows	A list of **window-*type*** elements.	Elements:	**window-*type***, where *type* can be one of the following types: • **aggregate** • **compute** • **copy** • **filter** • **join** • **pattern** • **procedural** • **source** • **union**
		Attributes:	None

		Example:	`<windows>`
			`  <window-source name='factInput'`
			`  id='win_001'...</window-source>`
			`  <window-source name='dimensionInput'`
			`  id='win_002'...</window-source>`
			`  <window-join name='joinedInput'`
			`  id='win_003'...</window-join>`
			`</windows>`

Element	Description	Details	
window-aggregate	An aggregation window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** **output** [**expr-initialize**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Example:	See "XML Examples of Aggregate Windows" on page 74.

Element	Description	Details	
window-compute	A compute window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** **output** [**expr-initialize**] [**context-plugin**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Example:	See "Window-Compute Examples" on page 69.

Element	Description	Details	
window-copy	A copy window.	Elements:	[**splitter-expr**] |
			[**spliter-plug**]
			[**retention**]
			[**connectors**]
		Attributes:	**name=*string***
			Window name
			[**id=*string***]
			Window ID
			[**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**]
			Index type for the window.
			pubsub= none | auto | manual
			Publish/subscribe mode for the window.
			[**pubsub-index=*value***]
			Publish/subscribe index value.
			[**insert-only**]
			[**output-insert-only**]
			[**collapse-updates**]
		Example:	See "Window-Copy Example" on page 69.

Element	Description	Details	
window-filter	A filter window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **expression** | **plugin** [**expr-initialize**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Example:	`<window-filter name='filterSrc' id='filterSrc_filter' index='pi_EMPTY'> <expression> not(isnull(src) or isnull(customerURI)) </expression> </window-filter>`

Element	Description	Details	
window-join	A join window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **join** **output** [**expr-initialize**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**] Index type for the window. **pubsub= none | auto | manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Example:	See "Window-Join Examples" on page 70.

Element	Description	Details	
window-pattern	A pattern window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** **patterns** [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Examples:	See "XML Pattern Window Examples" on page 93.

Element	Description	Details	
window-procedural	A procedural window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** [**connectors**] [**context-plugin**] [**plugin**] [**ds2-code**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**] Index type for the window. **pubsub= none | auto | manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Example:	See "XML Examples of Procedural Windows" on page 103.

Element	Description	Details	
window-source	A source window.	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** [**retention**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**]
		Example:	See "Window-Source Example" on page 69.

Element	Description	Details	
window-textcontext	A window that enables the abstraction of classified terms from an unstructured string field.	Elements:	[**splitter-expr**] | [**spliter-plug**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**] Index type for the window. **pubsub= none | auto | manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] **liti-fields="*list*"** Comma-separated list of LITI files. **text-field="*fieldname*"** Name of the field in the input window that contains the text to analyze.
		Example:	**<window-textcontext name='TWEET_ANALYSIS' id='tweet_analysis' liti-files='/opt/liti/english_tweets.liti,/opt/liti/cs_slang.liti' text-field='tweetBody>**

Element	Description	Details	
window-union	A union window.	Elements:	[**splitter-expr**] | [**spliter-plug**] [**connectors**]
		Attributes:	**name=*string*** Window name [**id=*string***] Window ID [**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**] Index type for the window. **pubsub= none | auto | manual** Publish/subscribe mode for the window. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] [**strict = true | false**] When true, two inserts on the same key from different inputs fail. When false, all input events that are Inserts are treated as Upserts. In this case, two Inserts for the same key on different input windows work, but you cannot determine which one is applied first.
		Example:	**<window-union name='combinedElementAttributeStats' id='combinedElementAttributeStats' strict='false' pubsub='true'> </window-union>**

XML Window Examples

Window-Source Example

```
<window-source name='logReadings' id='logReadings_src' index='pi_EMPTY'>
   <schema>
      <fields>
         <field name='ID' type='int32' key='true'/>
         <field name='udate' type='date'/>
         <field name='hostname' type='string'/>
         <field name='message' type='string'/>
      </fields>
   </schema>
   <connectors>
       <connector class='fs'>
         <properties>
             <property name='type'>pub</property>
             <property name='fstype'>syslog</property>
             <property name='fsname'>data/2013-10-11-clean.log.bi</property>
             <property name='growinginputfile'>true</property>
             <property name='transactional'>true</property>
             <property name='blocksize'>128</property>
         </properties>
        </connector>
    </connectors>
</window-source>
```

Window-Copy Example

```
<window-copy name='SrcIPCollision' id='SrcIPCollision_copy'>
   <retention type='bycount_sliding'>4096</retention>
   <connectors>
      <connector class='fs'>
        <properties>
           <property name='type'>sub</property>
           <property name='fstype'>csv</property>
           <property name='fsname'>SrcIPCollision.csv</property>
           <property name='snapshot'>true</property>
        </properties>
      </connector>
   </connectors>
</window-copy>
```

Window-Compute Examples

```
<window-compute name='computeWindow' id='compute_01'>
  <expr-initialize type='int32'>integer counter counter=0</expr-initialize>
  <schema>
    <fields>
      <field name='ID' type='int32' key='true'/>
      <field name='name' type='string'/>
```

```
      <field name='city' type='string'/>
      <field name='match' type='int32'/>
    </fields>
   </schema>
   <output>
      <field-expr>name</field-expr>
      <field-expr>city</field-expr>
      <field-expr>counter=counter+1 return counter</field-expr>
   </output>
</window-compute>

<window-compute name='compute' id='compute_01'>
  <schema>
    <fields>
      <field name='Id' type='int32' key='true'/>
      <field name='rowid' type='int64'/>
    </fields>
   </schema>
   <output>
      <field-plug plugin='libmethod' function='rowid'/>
   </output>
</window-compute>
```

Window-Join Examples

```
<window-join name='join_w' id='join_w'>
  <join type=type='leftouter' left='src1' right='src2'>
    <conditions>
       <fields left='element' right='element'/>
       <fields left='attribute' right='attribute'/>
    </conditions>
  </join>
  <output>
    <field-selection name='ID' source='l_ID'/>
    <field-selection name='element' source='l_element'/>
    <field-selection name='attribute' source='l_attribute'/>
    <field-selection name='value' source='l_value'/>
    <field-selection name='timestamp' source='l_timestamp'/>
    <field-selection name='status' source='l_status'/>
    <field-selection name='switch' source='r_switch'/>
  </output>
</window-join>

<window-join name='AddTraderName' id='AddTraderName_0004'>
  <join type="leftouter" left="LargeTrades_0003" right='Traders_0002' >
    <conditions>
      <fields left='traderID' right='ID' />
    </conditions>
  </join>
  <output>
    <field-expr name='security' type='string'>l_security</field-expr>
    <field-expr name='quantity' type='int32'>l_quantity</field-expr>
    <field-expr name='price' type='double'>100.0*l_price</field-expr>
    <field-expr name='traderID' type='int64'>l_traderID</field-expr>
    <field-expr name='time' type='stamp'>l_time</field-expr>
    <field-expr name='name' type='string'>r_name</field-expr>
```

```
  </output>
</window-join>
```

Chapter 5

Creating Aggregate Windows

Overview to Aggregate Windows

Aggregate windows are similar to compute windows in that non-key fields are computed. However, key fields are specified, and not inherited from the input window. Key fields must correspond to existing fields in the input event. Incoming events are placed into aggregate groups with each event in a group that has identical values for the specified key fields.

For example, suppose that the following schema is specified for input events:

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

Suppose that the aggregate window has a schema specified as the following:

```
"symbol*:string,totalQuant:int32,maxPrice:double"
```

When events arrive in the aggregate window, they are placed into aggregate groups based on the value of the **`symbol`** field. Aggregate field calculation functions (written in C++) or expressions that are registered to the aggregate window must appear in the non-key fields, in this example **`totalQuant`** and **`maxPrice`**. Either expressions or functions must be used for all of the non-key fields. They cannot be mixed. The functions or expressions are called with a group of events as one of their arguments every time a new event comes in and modifies one or more groups.

These groups are internally maintained in the **`dfESPwindow_aggregate`** class as **`dfESPgroupstate`** objects. Each group is collapsed every time that a new event is added or removed from a group by running the specified aggregate functions or expressions on all non-key fields. The purpose of the aggregate window is to produce one aggregated event per group.

Flow of Operations

The flow of operations while processing an aggregate window is as follows:

1. An event, **E** arrives and the appropriate group is found, called **G**. This is done by looking at the values in the incoming event that correspond to the key fields in the aggregate window
2. The event **E** is merged into the group **G**. The key of the output event is formed from the group-by fields of **G**.
3. Each non-key field of the output schema is computed by calling an aggregate function with the group **G** as input. The aggregate function computes a scalar value for the corresponding non-key field.
4. The correct output event is generated and output.

XML Examples of Aggregate Windows

```
<window-aggregate name='readingsPerElementAndAttribute' id='readingsPerElementAndAttribute' >
  <schema>
    <fields>
      <field name='element' type='string' key='true'/>
      <field name='attribute' type='string' key='true'/>
      <field name='value' type='double'/>
      <field name='timestamp' type='stamp'/>
      <field name='elementReadingCount' type='int64'/>
      <field name='startTime' type='stamp'/>
      <field name='endTime' type='stamp'/>
      <field name='valueAve' type='double'/>
      <field name='valueMin' type='double'/>
      <field name='valueMax' type='double'/>
      <field name='valueStd' type='double'/>
    </fields>
   </schema>
   <output>
      <field-expr>ESP_aLast(value)</field-expr>
      <field-expr>ESP_aLast(timestamp)</field-expr>
      <field-expr>ESP_aCountOpcodes(1)</field-expr>
      <field-expr>ESP_aFirst(timestamp)</field-expr>
      <field-expr>ESP_aLast(timestamp)</field-expr>
      <field-expr>ESP_aAve(value)</field-expr>
      <field-expr>ESP_aMin(value)</field-expr>
      <field-expr>ESP_aMax(value)</field-expr>
      <field-expr>ESP_aStd(value)</field-expr>
    </output>
</window-aggregate>
```

```
<window-aggregate name='aw_01' id='aggregateWindow_01'>
  <schema>
    <fields>
       <field name='symbol' type='string' key='true'/>
       <field name='totalQuant' type='int32'/>
       <field name='maxPrice' type='double'/>
    </fields>
   </schema>
   <output>
     <field-plug function='summationAggr' plugin='libmethod' additive='true'/>
```

```
      <field-plug function='maximumAggr' plugin='libmethod' additive='false'/>
    </output>
    <connectors>
       <connector class='fs'>
          <properties>
             <property name='type'>sub</property>
             <property name='fstype'>csv</property>
             <property name='fsname'>aggregate.csv</property>
             <property name='snapshot'>true</property>
          </properties>
       </connector>
     </connectors>
</window-aggregate>
```

Chapter 6

Creating Join Windows

Overview to Join Windows

A join window takes two input windows and a join type. For example,

- left outer window
- right outer window
- inner join
- full outer join

A join window takes a set of join constraints and a non-key field signature string. It also takes one of the following for the calculation of the join non-key fields when new input events arrive:

- a join selection string that is a one-to-one mapping of input fields to join fields
- field calculation expressions
- field calculation functions

A join window produces a single output stream of joined events. Because the SAS Event Stream Processing Engine is based on primary keys and supports Inserts, Updates, and Deletes, there are some restrictions placed on the types of joins that can be used.

The left window is the first window added as a connecting edge to the join window. The second window added as a connecting edge is the right window.

Examples of Join Windows

The following example shows a left outer join. The left window processes fact events and the right window processes dimension events.

```
left input schema: "ID*:int32,symbol:string,price:double,quantity:int32,
                           traderID:int32"

right input schema: "tID*:int32,name:string"
```

If **sw_01** is the window identifier for the left input window and **sw_02** is the window identifier for the right input window, your code would look like this:

```
dfESPwindow_join *jw;
jw = cq->newWindow_join("myJoinWindow", dfESPwindow_join::jt_LEFTOUTER,
                        edm, dfESPindextypes::pi_RBTREE);
jw-> setJoinConditions ("l_ID==r_tID");
jw->setJoinSelections("l_symbol,l_price,l_traderID,r_name");
jw->setFieldSignatures("sym:string,price:double,tID:int32,
     traderName:string");
```

Note the following:

- Join constraints take the following form. They specify what fields from the left and right events are used to generate matches.

  "l_*fieldname*=r_*fieldname*, ...,l_*fieldname*=r_*fieldname*"

- Join selection takes the following form. It specifies the list of non-key fields that are included in the events generated by the join window.

  "{l|r}_*fieldname*, ...{l|r}_*fieldname*"

- Field signatures take the following form. They specify the names and types of the non-key fields of the output events. The types can be inferred from the fields specified in the join selection. However, when using expressions or user-written functions (in C++), the type specification cannot be inferred, so it is required:

  "*fieldname*:*fieldtype*, ..., *fieldname*:*fieldtype*

When you use non-key field calculation expressions, your code looks like this:

```
dfESPwindow_join *jw;
jw = cq->newWindow_join("myJoinWindow", dfESPwindow_join::jt_LEFTOUTER,
                        edm, dfESPindextypes::pi_RBTREE);
jw->setJoinConditions("l_ID==r_tID");
jw->addNonKeyFieldCalc("l_symbol");
jw->addNonKeyFieldCalc("l_price");
jw->addNonKeyFieldCalc("l_traderID");
jw->addNonKeyFieldCalc("r_name");
jw->setFieldSignatures("sym:string,price:double,tID:int32,
     traderName:string");
```

This shows one-to-one mapping of input fields to join non-key fields. You can use calculation expressions and functions to generate the non-key join fields using arbitrarily complex combinations of the input fields.

Understanding Join Processing

For allowed one-to-Many and Many-to one joins, a change to the FACT table allows immediate lookup of the matching record in the dimension table through the primary index. This is possible because all key values of the dimension table are mapped in the join constraints. In fact, that is the definition of a dimension table. However, a change to the dimension table does not include a single primary key for a matching record in the FACT table. This illustrates the many-to-one nature of the join. In these cases the default mechanism to find the set of matching records in the FACT table is to perform a table scan and look for matches.

For small changes to the dimension table, this strategy is fine when no additional secondary index maintenance takes place. Hence, the join processing can be optimized. This is a common case. For example, the dimension table is a static lookup table that can be pre-loaded, and all subsequent changes happen on the FACT table.

However, there are cases where a large number of changes can be completed for the dimension table. One example is a source window that feeds an aggregation to produce statistics. The aggregation is joined back to the original source window to augment the original events with the statistics. In a case like this, the table scan to find the matching records for a change to the dimension table occurs with each change to the source window. This method is slow to invalidate the use case without further optimizations.

To solve this performance issue, you can implement automatic secondary index generation. Set the secondary parameter to **`true`** when constructing a new **`dfESPwindow_join`** instance. This causes a secondary index to be automatically generated and maintained when the join type involves a dimension table. When changes are made to the dimension table, table scans are eliminated. Therefore, performance is much faster.

You encounter a slight performance penalty when running with secondary indexes activated because the index needs to be maintained with every update to the FACT table. However, this secondary index maintenance is insignificant compared with elimination of table scans. With large tables, using secondary indexes often afford time savings of two orders of magnitude.

To turn on secondary index maintenance as stated, specify true as the final argument to the join constructor as follows:

```
jw = cq->newWindow_join("myJoinWindow", dfESPwindow_join::jt_LEFTOUTER,
                        edm, dfESPindextypes::pi_RBTREE, true);
```

Suppose you are doing a left outer or right outer join. You do not want to regenerate the entire table if the lookup window (dimension window) changes. In this case, you can specify **`true`** for the no-regenerate parameter. The no-regenerate parameter is an optional final parameter to **`newWindow_join()`** call, and when not present, defaults to false.

When the no-regenerate parameter is true, the join window runs in a highly optimized mode, where it does not need to keep its own index for the fact table. This saves memory, and also permits a change to the dimension side of the join to produce no output from the join window.

Understanding Join Constraints

The following table specifies where each Join type is permitted. The following standard notation is used: 1-1 for one-to-one, M-1 for many-to-one, and 1-M for one-to-many.

Join Type	Left Outer	Right Outer	Inner
1-1	Allowed	Allowed	Allowed
M-1	Allowed	Not Allowed	Allowed
1-M	Not Allowed	Allowed	Allowed

Note: Many-to-many (M-M) joins are strictly prohibited.

Classify left and right windows as dimension or fact windows as follows:

- dimension window - all keys of the window are represented in the join constraints.
- fact window - all keys of the window are not represented in the join constraints.

The following table specifies how keys of the Join window are computed given the type of window on the left and the right sides of the join. It also specifies the permitted Join types for that condition:

Join Type	Left Side	Right Side	Join Constraints	Permitted Join Types
1-1	Dimension	Dimension	All keys on left and right participate in the constraint set.	• For Left Outer, choose Keys of Left window. • For Right Outer, choose Keys from Right window. • For Inner, choose Keys from Left window (although keys of Right window would also work). • For Full Outer, choose keys of the Left window.

Join Type	Left Side	Right Side	Join Constraints	Permitted Join Types
M-1	Fact	Dimension	All keys on right and not all keys on left participate in the constraint set.	• For Left Outer, choose keys of Left window (Right window is lookup table). • Right Outer is not Allowed. • For Inner, choose keys from Left window.
1-M	Dimension	Fact	All keys on the left and not all keys on the right participate in the constraint set.	• Left Outer is not allowed. • For Right Outer, choose keys of Right window (Left windows is lookup table). • For Inner, choose keys from Right window.
M-M	Fact	Fact	Neither side has all keys participating in the constraint set.	Not allowed.

Creating Empty Index Joins

Suppose there is a lookup table and an insert-only fact stream. You want to match the fact stream against the lookup table (generating an Insert) and pass the stream out of the join for further processing. In this case the join does not need to store any fact data. Because no fact data is stored, any changes to the dimension data affect only subsequent rows. The changes cannot go back through existing fact data (because the join is stateless) and issue updates. The no-regenerate property must be enabled to ensure that the join does not try to go back through existing data.

Suppose there is a join of type LEFT_OUTER or RIGHT_OUTER. The index type is set to pi_EMPTY, rendering a stateless join window. The no-regenerate flag is set to TRUE. This is as lightweight a join as possible. The only retained data in the join is a local reference-counted copy of the dimensions table data. This copy is used to perform lookups as the fact data flows into, and then out of, the join.

Chapter 7

Creating Pattern Windows

Overview of Pattern Windows

A pattern is an algebraic expression using the logical operators AND, OR, NOT, and/or FBY (followed by) and expressions of interest. Expressions of interest are a WHERE-clause expression that include fields from an incoming event stream.

Here is a simple example:

```
expression of interest e1: (a==10)
expression of interest e2: (b==5 and cost>100.00)
pattern: e1 fby e2
pattern tree view:
               fby
               / \
              /   \
             e1    e2
```

Here is a slightly more complex example:

```
expression of interest e1: (a==10 and op=="insert")
expression of interest e2: (b==5 and cost>100.00)
expression of interest e3: (volume>1000)
pattern: e3 and (e1 fby e2)
pattern tree view:
              and
              / \
             /   \
            e3   fby
                 / \
                /   \
               e1    e2
```

When you create a pattern window, you do the following:

- declare a list of events of interest (EOIs)
- connect those events into an expression that use logical operators and optional temporal conditions

For example, you can have a combination of EOIs that occurs or does not occur within a specified period.

Time for events can be driven in real time or can be defined by a date-time or timestamp field. This field appears in the schema that is associated with the window that feeds the pattern window. In the latter case, you must ensure that incoming events are in order with respect to the field-based date-time or timestamp.

You can define multiple patterns within a pattern window. Each pattern typically has multiple events of interest, possibly from multiple windows or just one input window.

Specify EOIs by providing the following:

- a pointer for the window from where the event is coming
- a string name for the EOI
- a WHERE clause on the fields of the event, which can include a number of unification variables (bindings)

Suppose there is a single window that feeds a pattern window, and the associated schema is as follows:

```
ID*:int32,symbol:string,price:double,buy:int32,tradeTime:date
```

Suppose further that are two EOIs and that their relationship is temporal. You are interested in one event followed by the other within some period of time. This is depicted in the following code segment:

```
// Someone buys (or sells IBM) at price > 100.00
//     followed within 5 seconds of selling (or buying) SUN at price
//       > 25.00
dfESPpatternUtils::patternNode *l,*r, *f;
l = p_01->addEvent(sw_01, "e1",
                   "((symbol==\"IBM\") and (price > 100.00)
                   and (b == buy))");
r = p_01->addEvent(sw_01, "e2",
                   "((symbol==\"SUN\") and (price > 25.000)
                   and (b == buy))");
f = p_01->fby_op(l, r, 5000000); // note 5,000,000 microseconds
     = 5 seconds
```

Here there are two EOIs, **`l`** and **`r`**. The beginning of the WHERE clauses is standard: **`symbol==`*`constant`*` and price>`*`constant`**. The last part of each WHERE clause is where event unification occurs.

Because **`b`** is not a field in the incoming event, it is a free variable that is bound when an event arrives. It matches the first portion of the WHERE clause for event **`l`** (for example, an event for IBM with price > 100.00.) In this case, **`b`** is set to the value of the field **`buy`** in the matched event. This value of **`b`** is then used in evaluating the WHERE clause for subsequent events that are candidates for matching the second event of interest **`r`**. The added unification clause **`and (b == buy)`** in each event of interest ensures that the same value for the field **`buy`** appears in both matching events.

The FBY operator is sequential in nature. A single event cannot match on both sides. The left side must be the first to match on an event, and then a subsequent event could match on the right side.

The AND and OR operator are not sequential. Any incoming event can match EOIs on either side of the operator and for the first matching EOI causes the variable bindings. Take special care in this case, as this is rarely what you intend when you write a pattern.

For example, suppose that the incoming schema is as defined previously and you define the following pattern:

```
// Someone buys or sells IBM at price > 100.00 and also
//      buys or sells IBM at a price > 102.00 within 5 seconds.
l = p_01->addEvent(sw_01, "e1",
                   "((symbol==\"IBM\") and (price > 100.00)");
r = p_01->addEvent(sw_01, "e2", "((symbol==\"IBM\") and (price
                                   > 102.00)");
f = p_01->and_op(l, r, 5000000); // note 5,000,000 microseconds
     = 5 seconds
```

Now suppose an event comes into the window where symbol is "IBM" and price is "102.1". Because this is an AND operator, no inherent sequencing is involved, and the WHERE clause is satisfied for both sides of the "and" by the single input event. Thus, the pattern becomes true, and event **`l`** is the same as event **`r`**. This is probably not what you intended. Therefore, you can make slight changes to the pattern as follows:

```
// Someone buys (or sells IBM) at price > 100.00 and <= 102.00
// and also buys or selld IBS) at a price > 102.00 within 5 seconds.
l = p_01->addEvent(sw_01, "e1",
                   "(symbol==\"IBM\") and (price > 100.00) and
                   (price <= 102.00)");
r = p_01->addEvent(sw_01, "e2", "(symbol==\"IBM\") and (price
     > 102.00)");
f = p_01->and_op(l, r, 5000000); // note 5,000,000 microseconds
     = 5 seconds
```

After you make these changes, the price clauses in the two WHERE clauses disambiguate the events so that a single event cannot match both sides. This requires two unique events for the pattern match to occur.

Suppose that you specify a temporal condition for an AND operator such that event **`l`** and event **`r`** must occur within five seconds of one another. In that case, temporal conditions for each of the events are optional.

State Definitions for Operator Trees

Operator trees can have one of the following states:

- initial - no events have been applied to the tree
- waiting - an event has been applied causing a state change, but the left (and right, if applicable) arguments do not yet permit the tree to evaluate to TRUE or FALSE
- TRUE or FALSE - sufficient events have been applied for the tree to evaluate to a logical Boolean value

The state value of an operator sub-tree can be FIXED or not-FIXED. When the state value is FIXED, no further events should be applied to it. When the state value is not-FIXED, the state value could change based on application of an event. New events should be applied to the sub-tree.

When a pattern instance fails to emit a match and destroys itself, it folds. The instance is freed and removed from the active pattern instance list. When the top-level tree in a

pattern instance (the root node) becomes FALSE, the pattern folds. When it becomes TRUE the pattern emits a match and destroys itself.

An operator tree (*OPT*) is a tree of operators and EOIs. Given that *EO* refers to an event of interest or operator tree (*EOI* | *OPT*):

not *EOI*
becomes TRUE and FIXED or FALSE and FIXED on the application of a single event. It becomes TRUE if the event is applied it does not satisfy the event of interest, and FALSE if it does

not *OPT*
is a Boolean negation. This remains in the waiting state until *OPT* evaluates to TRUE or FALSE. Then it performs the logical negation. It only becomes FIXED when *OPT* becomes FIXED

notoccur *EOI*
becomes TRUE on application of an event that does not satisfy the *EOI*, but it is not marked FIXED. This implies that more events can be applied to it. As soon as it sees an event that matches the *EOI*, it becomes FALSE and FIXED

notoccur *OPT*
this is not allowed

EO* or *EO
is an event that is always applied to all non-FIXED sub-trees. It becomes TRUE when one of its two sub-trees become TRUE. It becomes FALSE when both of the sub-trees becomes FASLE. It is FIXED when one of its sub-trees is TRUE and FIXED, or both of its sub-trees are FALSE and not FIXED

EO* and *EO
is an event that is always applied to all non-FIXED sub-trees. It becomes TRUE when both of its two sub-trees become TRUE. It becomes FALSE when one of the sub-trees becomes FALSE. It is FIXED when one of its sub-trees is FALSE and FIXED or both of its sub-trees are TRUE and FIXED

EO* FBY *EO
attempts to complete the left hand side (LHS) with the minimal number of event applications before applying events to the right hand side (RHS). The apply rule is as follows:

- If the LHS is not TRUE or FALSE, apply event to the LHS until it become TRUE or FALSE.
- If the LHS becomes FALSE, set the followed by state to FALSE and become FIXED.
- If the LHS becomes TRUE, apply all further events to the RHS until the RHS becomes TRUE or FALSE. If the RHS becomes FALSE, set the **FBY** state to FALSE and FIXED, if it becomes TRUE set the **FBY** state toTRUE and FIXED.

This algorithm seeks the minimal length sequence of events that completes an **FBY** pattern.

Sample operator trees and events	Description
`(a fby b)`	Detect a, ..., b, where ... can be any sequence.
`( a fby ( (notoccur c) and b ) )`	Detect a, ..., b: but there can be no c between a and b.

Sample operator trees and events	Description
`( a fby (not c) ) fby (not (not b))`	Detect a, X, b: when X cannot be c.
`( ( (a fby b) fby (c fby d) ) and (notocc`	Detect a, ..., b, ..., c, ..., d : but k does not occur anywhere in the sequence.
`a fby (notoccur(c) and b)`	Detect an FBY b with no occurrences of c in the sequence.
`(not not a) fby (not not b)`	Detect an FBY b directly, with nothing between a and b.
`a fby (b fby ((notoccur c) and d))`	Detect a ... b ... d, with no occurrences of c between a and b.
`(notoccur c) and (a fby (b fby d))`	Detect a ... b ... d, with no occurrences of c anywhere.

Restrictions on Patterns

The following restrictions apply to patterns that you define in pattern windows:

- An event of interest should be used in only one position of the operator tree. For example, the following code would return an error:

```
// Someone buys (or sells) IBM at price > 100.00
//      followed within 5 seconds of selling (or buying)
//      SUN at price > 25.00 or someone buys (or sells)
//      SUN at price > 25.00 followed within 5 seconds
//            of selling (or buying) IBM at price > 100.00
//
dfESPpatternUtils::patternNode *l,*r, *lp, *rp, *fp;
l = p_01->addEvent(sw_01, "e1",
                   "((symbol==\"IBM\") and (price > 100.00)
                   and (b == buy))");
r = p_01->addEvent(sw_01, "e2", "((symbol==\"SUN\") and
                   (price > 25.000) and (b == buy))");
lp = p_01->fby_op(l, r, 5000000);   // note microseconds
rp = p_01->fby_op(r, l, 5000000);   // note microseconds
fp = p_01->or_op(lp, rp, 5000000);
```

  To obtain the desired result, you need four events of interest as follows:

```
dfESPpatternUtils::patternNode *l0,*r0, *l1, *r1, *lp, *rp, *fp;
l0 = p_01->addEvent(sw_01, "e1", "((symbol==\"IBM\") and
                                  (price > 100.00) and (b == buy))");
r0 = p_01->addEvent(sw_01, "e2", "((symbol==\"SUN\") and
                                  (price > 25.000) and (b == buy))");
l1 = p_01->addEvent(sw_01, "e3", "((symbol==\"IBM\") and
                                  (price > 100.00) and (b == buy))");
r1 = p_01->addEvent(sw_01, "e4", "((symbol==\"SUN\") and
                                  (price > 25.000) and (b == buy))");
```

```
lp = p_01->fby_op(l0, r0, 5000000);   // note microseconds
rp = p_01->fby_op(l1, r1, 5000000);   // note microseconds
fp = p_01->or_op(lp, rp, 5000000);
```

- Pattern windows work only on Insert events.

  If there might be an input window generating updates or deletions, then you must place a procedural window between the input window and the pattern window. The procedural window then filters out or transforms non-insert data to insert data.

  Patterns also generate only Inserts. The events that are generated by pattern windows are indications that a pattern has successfully detected the sequence of events that they were defined to detect. The schema of a pattern consists of a monotonically increasing pattern HIT count in addition to the non-key fields that you specify from events of interest in the pattern.

  `dfESPpattern::addOutputField() and dfESPpattern::addOutputExpression()`

- When defining the WHERE clause expression for pattern events of interests, binding variables must always be on the left side of the comparison (like **`bindvar == field`**) and cannot be manipulated.

  For example, the following **`addEvent`** statement would be flagged as invalid:

  ```
  e1 = consec->addEvent(readingsWstats, "e1",
       "((vmin < aveVMIN) and (rCNT==MeterReadingCnt) and (mID==meterID))");
  e2 = consec->addEvent(readingsWstats, "e2",
       "((mID==meterID) and (rCNT+1==MeterReadingCnt) and (vmin < aveVMIN))");
  op1 = consec->fby_op(e1, e2,28800000001);
  ```

  Consider the WHERE clause in **`e1`**. It is the first event of interest to match because the operator between these events is a followed-by. It ensures that event field **`vmin`** is less than field **`aveVMIN`**. When this is true, it binds the variable **`rCNT`** to the current meter reading count and binds the variable **`mID`** to the **`meterID`** field.

  Now consider **`e2`**. Ensure the following:

  - the **`meterID`** is the same for both events
  - the meter readings are consecutive based on the **`meterReadingCnt`**
  - **`vmin`** for the second event is less than aveVMIN

  The error in this expression is that it checked whether the meter readings were consecutive by increasing the **`rCNT`** variable by 1 and comparing that against the current meter reading. Variables cannot be manipulated. Instead, you confine manipulation to the right side of the comparison to keep the variable clean.

  The following code shows the correct way to accomplish this check. You want to make sure that meter readings are consecutive (given that you are decrementing the meter reading field of the current event, rather than incrementing the variable).

  ```
  e1 = consec->addEvent(readingsWstats, "e1",
       "((vmin < aveVMIN) and (rCNT==MeterReadingCnt) and (mID==meterID))");
  e2 = consec->addEvent(readingsWstats, "e2",
       "((mID==meterID) and (rCNT==MeterReadingCnt-1) and (vmin < aveVMIN))");
  op1 = consec->fby_op(e1, e2,28800000001);
  ```

Using Stateless Pattern Windows

Pattern windows are insert-only with respect to both their input windows and the output that they produce. The output of a pattern window is a monotonically increasing integer ID that represents the number of patterns found in the pattern window. The ID is followed by an arbitrary number of non-key fields assembled from the fields of the events of interest for the pattern. Because both the input and output of a pattern window are unbounded and insert-only, they are natural candidates for stateless windows (that is, windows with index type **pi_EMPTY**).

Pattern windows are automatically marked as insert-only. They reject records that are not iInserts. Thus, no problems are encountered when you use an index type of **pi_EMPTY** with pattern windows. If a source window feeds the pattern window, it needs to be explicitly told that it is insert-only, using the **dfESPwindow::setInsertOnly()** call. This causes the source window to reject any events with an opcode other than Insert, and permits an index type of **pi_EMPTY** to be used.

Stateless windows are efficient with respect to memory use. More than one billion events have been run through pattern detection scenarios such as this with only modest memory use (less than 500MB total memory).

```
Source Window [insert only, pi_EMPTY index] --> PatternWindow[insert only,
     pi_EMPTY index]
```

Pattern Window Examples

C++ Pattern Window Example

Here is a complete example of a simple pattern window. For more examples, refer to the packaged examples provided with the product.

```
#define MAXROW 1024
#include <iostream>

// Include class definitions for modeling objects.

#include "dfESPeventdepot_mem.h"
#include "dfESPwindow_source.h"
#include "dfESPwindow_pattern.h"
#include "dfESPevent.h"
#include "dfESPcontquery.h"
#include "dfESPengine.h"
#include "dfESPproject.h"

using namespace std;

// This is a simple callback function that can be registered for
//      a windows new event updates.
// It receives the schema of the events it is passed, and a set of
//      1 or more events bundled into a dfESPeventblock object. It
```

```
//      also has an optional context pointer for passing state
//      into this cbf.

void winSubscribeFunction(dfESPschema *os, dfESPeventblockPtr ob,
     void *cntx) {
int count = ob->getSize(); // Get the size of the event block.
if (count>0) {
    char buff[MAXROW+1];
    for (int i=0; i < count; i++) {
       ob->getData(i)->toStringCSV(os, (char *)buff, MAXROW);
       // Get the event as CSV.
       cout << buff << endl; // Print it
       if (ob->getData(i)->getOpcode() ==
           dfESPeventcodes::eo_UPDATEBLOCK)
           ++i; // skip the old record in the update block
     } //for
 } //if
}

int main(int argc, char *argv[]) {

// ------ BEGIN MODEL (CONTINUOUS QUERY DEFINITIONS) ------
// Create the single engine top level container which sets up
//      dfESP fundamental services such as licensing, logging,
//      pub/sub, and threading, ...
// Engines typically contain 1 or more project containers.
// @param argc the parameter count as passed into main.
// @param argv the paramter vector as passed into main.
//      Currently the dfESP library only looks for
//      -t <textfile.name> to to write its output and
//      -b <badevent.name>  to write any bad events
//      (events that failed to be applied to a window index).
// @param id the user supplied name of the engine.
// @param pubsub pub/sub enabled/disabled and port pair,
//      formed by calling static function
//      dfESPengine::pubsubServer().
// @param logLevel the lower threshold for displayed log
//      messages - default: dfESPLLTrace, @see
//      dfESPLoggingLevel
// @param logConfigFile a log4SAS configuration file -
//      default: log to stdout.
// @param licKeyFile a FQPN to a license file - default:
//       $DFESP_HOME/etc/license/esp.lic
// @return the dfESPengine instance.

dfESPengine *myEngine = dfESPengine::initialize(argc, argv,
                        "engine", pubsub_DISABLE);
if (myEngine == NULL) {
   cerr <<"Error: dfESPengine::initialize() failed using
           all framework defaults\n";
   return 1;
}

// Define the project, this is a container for one or more
//      continuous queries.
dfESPproject *project_01 = myEngine->newProject("project_01");
```

```
// Create a memory depot for the project to handle the
//      generation of primary indices and event storage;

dfESPeventdepot_mem* depot_01;
depot_01 =project_01->newEventdepot_mem("memDepot_01");

// Define a continuous query object. This is the first level
//      container for windows. It also contains the window
//      to window connectivity information.
dfESPcontquery *cq_01;
cq_01 = project_01->newContquery("contquery_01");

// Build the source window. We specify the window name, the
//      schema for events, the depot used to generate the
//      index and handle event storage, and the type of
//      primary index, in this case a red/black tree index.
dfESPwindow_source *sw_01;
sw_01 = cq_01->newWindow_source("sourceWindow_01", depot_01,
                                dfESPindextypes::pi_RBTREE,
                                dfESPstring("ID*:int32,symbol:string,price:double,
                                            buy:int32,tradeTime:date"));

dfESPwindow_pattern *pw_01;
pw_01 = cq_01->newWindow_pattern("patternWindow_01", depot_01,

// Create a new pattern
dfESPpattern* p_01 = pw_01->newPattern();
{       dfESPpatternUtils::patternNode *e1,*e2, *o1;
        // Pattern of interest: someone buys IBM at price >
        //      100.00 followed within 5 second of buying
        //      SUN at price > 25.00.
        e1 = p_01->addEvent(sw_01, "e1",
                            "((symbol==\"IBM\") and (price >
                              100.00) and (b == buy))");
        e2 = p_01->addEvent(sw_01, "e2",
                            "((symbol==\"SUN\") and (price >
                              25.000) and (b == buy))");
        o1 = p_01->fby_op(e1, e2, 5000000);
        // e1 fby e2 within 5 sec

        p_01->setPattern(o1); //set the pattern top of op tree

        // Setup the generated event for pattern matches.
        p_01->addOutputField("ID", e1);
        p_01->addOutputField("ID", e2);
        p_01->addTimeField(sw_01, "tradeTime");
        //set tradeTime field for the temporal check
  }

// Add the subscriber callback to the pattern window.
pw_01->addSubscriberCallback(winSubscribeFunction);

// Add the connectivity information to the continuous query.
//      This means sw_o1 --> pw_01
cq_01->addEdge(sw_01, pw_01);
```

```
// Define the project's thread pool size and start it.
// **Note** after we start the project here, we do not see
//       anything happen, as no data has yet been put
//      into the continuous query.
project_01->setNumThreads(2);
myEngine->startProjects();        //

// ------ END MODEL (CONTINUOUS QUERY DEFINITION) ------

// At this point the project is running in the background
//      using the defined thread pool. We'll use the main
//      thread that we are in to inject some data.
// Generate some test event data and inject it into the
//      source window.

dfESPptrVect<dfESPeventPtr> trans;
dfESPevent     *p;

p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,1,IBM,
    101.45,0,2011-07-20 16:09:01");
trans.push_back(p);

dfESPeventblockPtr ib = dfESPeventblock::newEventBlock
     (&trans, dfESPeventblock::ebt_TRANS);
trans.free();
project_01->injectData(cq_01, sw_01, ib);
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,2,IBM,
    101.45,1,2011-07-20 16:09:02");
trans.push_back(p);
ib = dfESPeventblock::newEventBlock(&trans,
                                    dfESPeventblock:
                                    :ebt_TRANS);
trans.free();
project_01->injectData(cq_01, sw_01, ib);

p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,3,SUN,
                   26.0,1,2011-07-20 16:09:04");
trans.push_back(p);
ib = dfESPeventblock::newEventBlock(&trans,
                                    dfESPeventblock:
                                    :ebt_TRANS);
trans.free();
project_01->injectData(cq_01, sw_01, ib);
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,4,SUN,
                   26.5,0,2011-07-20 16:09:05");
trans.push_back(p);
ib = dfESPeventblock::newEventBlock(&trans,
                                    dfESPeventblock:
                                    :ebt_TRANS);
trans.free();

project_01->injectData(cq_01, sw_01, ib);
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,5,IBM,
                   101.45,1,2011-07-20 16:09:08");
trans.push_back(p);
```

```
ib = dfESPeventblock::newEventBlock(&trans,
                                    dfESPeventblock:
                                    :ebt_TRANS);
trans.free();
project_01->injectData(cq_01, sw_01, ib);

project_01->quiesce(); // Wait until the system stops
//      processing before shutting down.

// Now shutdown.
myEngine->stopProjects();
// Stop project before shutting
//     down.
myEngine->shutdown();
return 0;
}
```

After you execute this code, you obtain these results:

```
I,N: 0,2,3
I,N: 1,1,4
```

XML Pattern Window Examples

```
<window-pattern name='front_running'>
  <schema-string>
   id*:int64,broker:int32,brokerName:string,typeFlg:int32,symbol:
       string,tstamp1:date,tstamp2:date,tstamp3:date,tradeId1:
       int32,tradeId2:int32,tradeId3:int32
  </schema-string>
  <patterns>
    <pattern>
      <events>
        <event name='e1'>((buysellflg==1) and (broker == buyer)
                          and (s == symbol) and (b == broker)
                          and (p == price))
        </event>
        <event name='e2'>((buysellflg==1) and (broker != buyer)
                          and (s == symbol) and (b == broker))
        </event>
        <event name='e3'><![CDATA[
          buysellflg==0) and (broker == seller)
          and (s == symbol) and (b == broker)
          and (p < price))]]></event>
      </events>
      <logic>fby(e1,e2,e3)</logic>
      <output>
       <field-selection name='broker' node='e1'/>
       <field-selection name='brokerName' node='e1'/>
       <field-expr>1</field-expr>
       <field-selection name='symbol' node='e1'/>
       <field-selection name='date' node='e1'/>
       <field-selection name='date' node='e2'/>
       <field-selection name='date' node='e3'/>
```

```
                        <field-selection name='id' node='e1'/>
                        <field-selection name='id' node='e2'/>
                        <field-selection name='id' node='e3'/>
                       </output>
                      </pattern>
                     <pattern>
                       <events>
                         <event name='e1'>((buysellflg==0) and (broker == seller)
                                          and (s == symbol) and (b == broker))
                         </event>
                         <event name='e2'>((buysellflg==0) and (broker != seller)
                                          and (s == symbol) and (b == broker))
                         </event>
                       </events>
                       <logic>fby(e1,e2)</logic>
                       <output>
                         <field-selection name='broker' node='e1'/>
                         <field-selection name='brokerName' node='e1'/>
                         <field-expr>2</field-expr>
                         <field-selection name='symbol' node='e1'/>
                         <field-selection name='date' node='e1'/>
                         <field-selection name='date' node='e2'/>
                         <field-expr> </field-expr>
                         <field-selection name='id' node='e1'/>
                         <field-selection name='id' node='e2'/>
                         <field-expr>0</field-expr>
                       </output>
                      </pattern>
                     </patterns>
                   </window-pattern>

<window-pattern name='sigma2Pattern_calc' id='sigma2Pattern_calc' index='pi_EMPTY' >
          <schema>
             <fields>
                <field name='alertID' type='int64' key='true'/>
                <field name='ID1' type='int64'/>
                <field name='element1' type='string'/>
                <field name='attribute1' type='string'/>
                <field name='timestamp1' type='stamp'/>
                <field name='value1' type='double'/>
                <field name='valueAve1' type='double'/>
                <field name='valueMin1' type='double'/>
                <field name='valueMax1' type='double'/>
                <field name='valueStd1' type='double'/>
                <field name='ID2' type='int64'/>
                <field name='element2' type='string'/>
                <field name='attribute2' type='string'/>
                <field name='timestamp2' type='stamp'/>
                <field name='value2' type='double'/>
                <field name='valueAve2' type='double'/>
                <field name='valueMin2' type='double'/>
                <field name='valueMax2' type='double'/>
                <field name='valueStd2' type='double'/>
             </fields>
          </schema>
          <patterns>
```

```
      <pattern index='element,attribute'>
         <events>
            <event source='original2stdDevCheck' name='e1'>
              (value&lt;(valueAve-2*valueStd))
                        and (r_cnt==elementReadingCount)
                        and (pid==element and aid==attribute)
            </event>
            <event source='original2stdDevCheck' name='e2'>
              (pid==element and aid==attribute)
               and (r_cnt&gt;=elementReadingCount-2)
               and (value&lt;(valueAve-2*valueStd))
            </event>
         </events>
         <logic>fby{18600000001 seconds}(e1, e2)</logic>
         <output>
            <field-selection name='sequence' node='e1'/>
            <field-selection name='element' node='e1'/>
            <field-selection name='attribute' node='e1'/>
            <field-selection name='timestamp' node='e1'/>
            <field-selection name='value' node='e1'/>
            <field-selection name='valueAve' node='e1'/>
            <field-selection name='valueMin' node='e1'/>
            <field-selection name='valueMax' node='e1'/>
            <field-selection name='valueStd' node='e1'/>
            <field-selection name='sequence' node='e2'/>
            <field-selection name='element' node='e2'/>
            <field-selection name='attribute' node='e2'/>
            <field-selection name='timestamp' node='e2'/>
            <field-selection name='value' node='e2'/>
            <field-selection name='valueAve' node='e2'/>
            <field-selection name='valueMin' node='e2'/>
            <field-selection name='valueMax' node='e2'/>
            <field-selection name='valueStd' node='e2'/>
         </output>
      </pattern>
   </patterns>
</window-pattern>
```

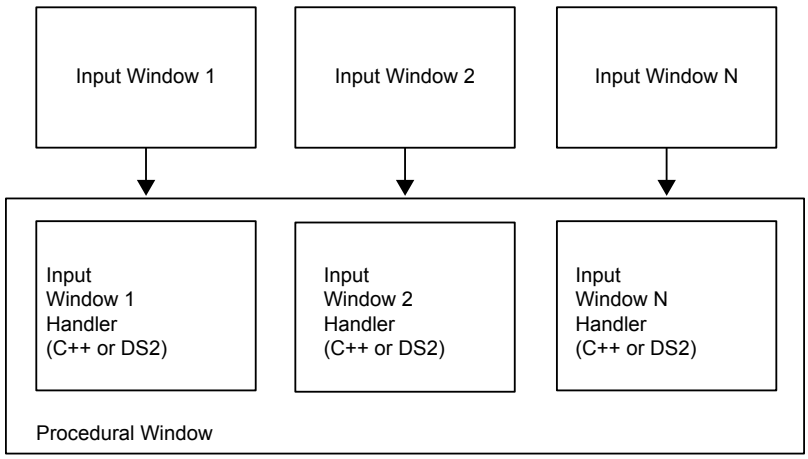
Chapter 8

Creating Procedural Windows

Overview to Procedural Windows

You can write procedural window input handlers in C++, DS2, or XML. When an input event arrives, the handler registered for the matching input window is called. The events produced by this handler function are output.

Figure 8.1 *Procedural Window with Input Handlers*



In order for the state of the procedural window to be shared across handlers, an instance-specific context object (such as **`dfESPpcontext`**) is passed to the handler function. Each handler has full access to what is in the context object. The handler can store data in this context for use by other handlers, or by itself during future invocations.

C++ Window Handlers

Here is an example of the signature of a procedural window handler written in C++.

```
typedef bool (*dfESPevent_func)(dfESPpcontext *pc,
              dfESPschema *is, dfESPeventPtr nep,
               dfESPeventPtr oep, dfESPschema *os,
              dfESPptrVect<dfESPeventPtr>&oe);
```

The procedural context is passed to the handler. The input schema, the new event, and the old event (in the case of an update) are passed to the handler when it is called. The final parameters are the schema of the output event (the structure of events that the procedural window produces) and a reference to a vector of output events. It is this vector where the handler needs to push its computed events.

Only one input window is defined, so define only one handler function and call it when a record arrives.

```
// This handler functions simple counts inserts, updates,
//      and deletes.
// It generates events of the form "1,#inserts,#updates,
//      #deletes"
//
bool opcodeCount(dfESPpcontext *mc, dfESPschema *is,
                    dfESPeventPtr nep, dfESPeventPtr oep,
                    dfESPschema *os, dfESPptrVect
                    <dfESPeventPtr>& oe) {

    derivedContext *ctx = (derivedContext *)mc;
    // Update the counts in the past context.
    switch (nep->getOpcode()) {
        case dfESPeventcodes::eo_INSERT:
           ctx->numInserts++;
           break;
        case dfESPeventcodes::eo_UPDATEBLOCK:
           ctx->numUpdates++;
           break;
        case dfESPeventcodes::eo_DELETE:
           ctx->numDeletes++;
           break;
      }

    // Build a vector of datavars, one per item in our output
    //      schema, which looks like: "ID*:int32,insertCount:
    //      int32,updateCount:int32,deleteCount:int32"

dfESPptrVect<dfESPdatavarPtr> vect;
os->buildEventDatavarVect(vect);

// Set the fields of the record that we are going to produce.

vect[0]->setI32(1); // We have a key of only 1, we keep updating one record.
vect[1]->setI32(ctx->numInserts);
vect[2]->setI32(ctx->numUpdates);
```

```
vect[3]->setI32(ctx->numDeletes);

// Build the output Event, and push it to the list of output
//      events.

dfESPeventPtr ev = new dfESPevent();
ev->buildEvent(os, vect, dfESPeventcodes::eo_UPSERT,
               dfESPeventcodes::ef_NORMAL);
oe.push_back(ev);

// Free space used in constructing output record.
vect.free();
return true;
```

The following example shows how this fits together in a procedural window:

```
dfESPproject *project_01;
project_01 = theEngine->newProject("project_01");

dfESPcontquery *cq_01;
cq_01 = project_01->newContquery("cq_01");

dfESPeventdepot_mem* depot;
depot = project_01->newEventdepot_mem("Depot_01");

dfESPstring source_sch = dfESPstring("ID*:int32,symbol:
                                     string,price:double");
dfESPstring procedural_sch = dfESPstring("ID*:int32,insertCount:
                                         int32,updateCount:int32,
                                         deleteCount:int32");

dfESPwindow_source *sw;
sw = cq_01->newWindow_source("source window", depot,
                             dfESPindextypes::pi_HASH,
                             source_sch);

dfESPwindow_procedural *pw;
pw = cq_01->newWindow_procedural("procedural window", depot,
                                 dfESPindextypes::pi_RBTREE,
                                 procedural_sch);

// Create our context, and register the input window and
//      handler.
//
derivedContext *mc = new derivedContext();
mc->registerMethod(sw, opcodeCount);

pw->registerMethodContext(mc);
```

Now whenever the procedural window sees an event from the source window (sw), it calls the handler opcodeCount with the context mc, and produces an output event.

DS2 Window Handlers

Overview of DS2 Window Handlers

When you write a procedural window handler in the DS2 programming language, the program is declared as a character string and set in the procedural windows context.

Here is a simple example:

```
char *DS2_program_01 =
        "ds2_options cdump;"
        "data esp.out;"
        "   dcl double cost;"
        "   method run();"
        "     set esp.in;"
        "     cost = price * quant;"
        "   end;"
        "enddata;"
```

The window handler is then added to the procedural window's context, before the context is registered with the procedural window proper.

```
/* declare the next context */
dfESPpcontext *pc_01 = new dfESPpcontext;
/* register the DS2 handler in the context */
pc_01->registerMethod_ds2(sw_01,  DS2_program_01);
/* register the context with the procedural window */
pw_01->registerMethodContext(pc_01);
```

All fields of the input window are seen as variables in DS2 programs, so can be used in calculations. The variable _opcode is also available and takes on the integer values **1** (Insert), **2** (Update), or **3**(Delete). The variables exported from the DS2 program are all the input variables plus any global variables declared. This set of variables is then filtered by the schema field names of the procedural window to form the output event.

General Structure of a DS2 Input Handler

DS2 input handlers use the following boilerplate definition:

```
ds2_options cdump;
    data esp.out;
       global_variable_declaration;   /* global variable block    */
        method run();
           set esp.in;
           computations; /* computational statements */
        end;
enddata;
```

Examples

```
input schema:
         "ID*:int32,symbol:string,size:int32,price:double"

   output (procedural schema):
```

```
        "ID*:int32,symbol:string,size:int32,price:double,cost:double"

    ds2_options cdump;
    data esp.out;
        dcl double cost;
        method run();
           set esp.in;
           cost = price * size;  /* compute the total cost */
        end;
    enddata;
```

Here is a procedural window with one input window that does no computation. It remaps the key structure, and omits some of the input fields:

```
input schema:
        "ID*:int32,symbol:string,size:int32,price:double,traderID:int32"

   output (procedural schema):
          "kID*:int64,symbol:string,cost:double"

    ds2_options cdump;
    data esp.out;
        dcl double cost;
        dcl bigint kID;
        method run();
           set esp.in;
           kID  = 1000000000*traderID;  /* put traderID in digits 10,11, ...*/
           kID  = kID + ID;             /* put ID in digits 0,1, ... 9      */
           cost = price * size;         /* compute the total cost */
        end;
    enddata;
```

Note: This DS2 code produces the following output: {ID, symbol, size, price, traderID, cost, kID}, which when filtered through the output schema is as follows: {kID, symbol, cost}

Here is a procedural window with one input window that augments an input event with a letter grade based on a numeric grade in the input:

```
input schema:
        "studentID*:int32,testNumber*:int32,testScore:double"

   output (procedural schema):
         "studentID*:int32,testNumber*:int32,testScore:double,testGrade:string"

    ds2_options cdump;
    data esp.out;
        dcl char(1) testGrade;
        method run();
           set esp.in;
                testGrade = select
                when (testScore >= 90) 'A'
                when (testScore >= 80) 'B'
                when (testScore >= 70) 'C'
                when (testScore >= 60) 'D'
                when (testScore >= 0)  'F'
        end;
```

```
    enddata;
```

Here is a procedural window with one input window that augments an input event with the timestamp of when it was processed by the DS2 Handler:

```
input schema:
        "ID*:int32,symbol:string,size:int32,price:double"

   output (procedural schema):
          "ID*:int32,symbol:string,cost:double,processedStamp:stamp"

    ds2_options cdump;
    data esp.out;
        method run();
           set esp.in;
           processedStamp = to_timestamp(datetime());
        end;
    enddata;
```

Here is a procedural window with one input window that filters events with an even ID. It produces two identical events (with different keys) for those events with an odd ID:

```
input schema:
        "ID*:int32,symbol:string,size:int32,price:double"

   output (procedural schema):
         "ID*:int32,symbol:string,size:int32,price:double"

    ds2_options cdump;
    data esp.out;
        method run();
           set esp.in;
           if MOD(ID, 2) = 0 then return;
           output;
           ID = ID + 1;
           output;
        end;
    enddata;
```

Given this input:

```
1,ibm,1000,100.1
    2,nec,2000,29.7
    3,ibm,2000,100.7
    4,apl,1000,300.2
```

The following output is produced:

```
1,ibm,1000,100.1
    2,ibm,1000,100.1
    3,ibm,2000,100.7
    4,ibm,2000,100.7
```

Event Stream Processor to DS2 Data Type Mappings and Conversions

The following mapping of event stream processor to DS2 data types is supported:

Event Stream Processor Data Type	DS2 Data Type
ESP_INT32	TKTS_INTEGER
ESP_INT64	TKTS_BIGINT
ESP_DOUBLE	TKTS_DOUBLE
ESP_TIMESTAMP/DATETIME	TKTS_TIMESTAMP/DATE/TIME
ESP_UTF8STR	TKTS_VARCHAR/CHAR

The ESP_MONEY data type is not supported.

Here is a conversion matrix. If a data type does not appear in the matrix (for example, NVarchar), conversion is not supported for it.

From/To	Integer	BigInt	Double	Date	Time	Timestamp	Char	Varchar
int32	x							
int64		x						
double			x					
datetime				x	x	x		
timestamp				x	x	x		
utf8str							x	x

XML Examples of Procedural Windows

You can write procedural windows in XML using the **window-procedural** element. For more information about this element, see "Dictionary of XML Elements" on page 44.

```
<window-procedural name='pw_01' id='proceduralWindow_01'>
  <context-plugin name='libmethod' function='get_derived_context'/>
  <schema>
    <fields>
      <field name='ID' type='int32' key='true' />
      <field name='insertCount' type='int32' />
      <field name='updateCount' type='int32'/>
```

```
      <field name='deleteCount' type='int32'/>
    </fields>
  </schema>
  <plugin source='sourceWindow_01' name='libmethod' function='countOpcodes'/>
    <connectors>
      <connector class='fs'>
        <properties>
          <property name='type'>sub</property>
          <property name='fstype'>csv</property>
          <property name='fsname'>procedural1.csv</property>
          <property name='snapshot'>true</property>
        </properties>
     </connector>
    </conncetors>
</window-procedural>

<window-procedural name='finalEmptySrcStats' id='finalEmptySrcStats_proc'>
  <schema>
    <fields>
      <field name='key' type='int32' key='true'/>
      <field name='srcNullCount' type='int32'/>
      <field name='srcNullCount' type='int32'/>
      <field name='srcZoneURINullCount' type='int32'/>
      <field name='cURINullCount' type='int32'/>
    </fields>
  </schema>
  <ds2-code source='emptySrcStats_compute'>
    <![CDATA{
       ds2_options cdump;
                data esp.out;
                  dcl integer key;
                    method run();
                       set esp.in;
                       key = 1;
                       _opcode = 4;
                    end;
                enddata;
     ]]>
  </ds2-code>
</window-procedural>
```

Chapter 9

Visualizing Event Streams

Overview to Event Visualization

The Streamviewer tool provided with SAS Event Stream Processing Engine enables you to subscribe to windows' event streams. You can run this tool, named **`dfesp_streamviewer.bat`** on most platforms, to query an engine's objects and containers down to the window level. You can subscribe to any window's event stream. You can find Streamviewer in the **`$DFESP_HOME/bin`** directory.

In the Microsoft Windows distribution of SAS Event Stream Processing Engine, an example shows how you can integrate the product with SAS/GRAPH for event stream visualization. You can find this example in the **`%DFESP_HOME%\src\graph_realtime`** directory. In order to use it, you must separately purchase SAS/GRAPH.

You can use SAS BI Dashboard for event stream visualization by having the dashboard subscribe to event streams of interest. You must install the SAS Event Stream Processing Engine plug-in to SAS BI Dashboard to enable this feature. For more information, see "Installing and Using the SAS BI Dashboard Plug-in" on page 107.

Using Streamviewer

Streamviewer subscribes to any window event stream and displays it. It can subscribe to event streams from a single engine or from different engines on separate machines. If the total number of events is bound, then you can view changes to values as they are updated. Otherwise, you see events scrolling through the viewer. You can set color thresholds based on any event field, which can be useful for alert counts.

Streamviewer is written in Java and it uses the publish/subscribe API. You can run it on a different system from the SAS Event Stream Processing Engine application as long as the system is on the same network.

Execute Streamviewer using the following command on UNIX and Linux:

$DFESP_HOME/bin/dfesp_streamviewer -u *URLs_to_ESP_windows*
-Y *window_height* -X *window_width*
-x *x_coordinate_of_window* -y *y_coordinate_of_window*
-d *millisecond_delay_before_subscribing* -h *usage*

Execute Streamviewer using the following command on Windows:

%DFESP_HOME%\bin\dfesp_streamviewer *URLs_to_ESP_windows*
-height *h* —width *w*
—initialX *x* -initialY *y*
-delay *d*

Note: **`/?`**, **`-?`**, **`-h`**, **`-H`**, **`/h`**, and **`/H`** all produce help.

After you start Streamviewer, there are two options under **File** ⇨ **New Window Subscribe** and **Exit**. You can choose **New Window Subscribe** any number of times. Each time you initiate the window subscribe option, a dialog box appears. In this dialog box, you can enter the host and port. Then Streamviewer queries the model for available projects, queries, and windows.

Dialog boxes then prompt you to enter the following:

- project name
- continuous query name
- window name
- optional rules file
- a toggle to select the initial snapshot before the continuous deltas and updates

If you subscribe to a system that is running events, then you want the snapshot that is the current state of the event stream. You can use a rules file for color thresholds based on a field expression.

Here is an example set of rules that are used with the **`broker_surv`** example provided in the installation:

```
frpbuyalerts,GREATERTHANEQUALTO,3,255:0:0
frpbuyalerts,LESSTHAN,3,255:255:0
frpbuyalerts,EQUALTO,0,0:255:0
frpsellalerts,GREATERTHANEQUALTO,3,255:0:0
frpsellalerts,LESSTHAN,3,255:255:0
frpsellalerts,EQUALTO,0,0:255:0
markopenalerts,GREATERTHANEQUALTO,3,255:0:0
markopenalerts,LESSTHAN,3,255:255:0
markopenalerts,EQUALTO,0,0:255:0
markclosealerts,GREATERTHANEQUALTO,3,255:0:0
markclosealerts,LESSTHAN,3,255:255:0
markclosealerts,EQUALTO,0,0:255:0
restrictedsalealerts,GREATERTHANEQUALTO,3,255:0:0
restrictedsalealerts,LESSTHAN,3,255:255:0
restrictedsalealerts,EQUALTO,0,0:255:0
totalalerts,GREATERTHANEQUALTO,5,255:0:0
totalalerts,LESSTHAN,5,255:255:0
totalalerts,LESSTHANEQUALTO,3,0:255:0
```

Each rule includes a field name, condition, field value, red value, blue value, and green value. If the field meets the condition, then that field is colored as specified (for example, 0-255 for each color component) in the color codes. Initially, each subscribe session has a tab. Use the **Open this** tab in a new window button to make it a separate window. You can also return to the main window using **File** ⇨ **Dock in Parent** . Finally, you can also run Streamviewer with parameters so that you can call it from a script without the Subscribe Initialize dialog box.

For information about these parameters, execute Streamviewer with the **`-h`** parameter.

Using SAS/GRAPH

SAS/GRAPH can be used to build both simple and complex combinations of graphs. You can integrate this tool with the SAS Event Stream Processing Engine using the publish/subscribe API. The Microsoft ActiveX control library included in SAS/GRAPH 9.3 or later is supported on Microsoft Windows.

Microsoft Windows (32-bit or 64-bit) contains a graph_realtime example that shows how to integrate SAS/GRAPH with the SAS Event Stream Processing Engine publish/subscribe API to subscribe to a SAS Event Stream Processing Engine and visualize events in graphs. This example uses a plot graph to continuously display total trades on the X-axis and throughput rate on the Y-axis based on the streaming events.

Installing and Using the SAS BI Dashboard Plug-in

Overview to Using the SAS BI Dashboard Plug-in

SAS Event Stream Processing Engine provides a plug-in to SAS BI Dashboard named **`$DFESP_HOME/bin/dfx-esp-bid-plugin.jar`**. After you install the plug-in, SAS BI Dashboard implements multiple subscribers to event stream processing windows. Each of these subscribers uses the same URL as a subscriber using the publish/subscribe API.

When subscribing to event stream processing publishers, SAS BI Dashboard grabs the most recent events from streams at regular intervals. You can configure this interval and the batch size through the dashboard. Because SAS BI Dashboard maintains an unindexed set of rows, the plug-in builds its own index of the subscribed window's events. The plug-in transparently handles Updates, Deletes, and Inserts.

Installing the SAS BI Dashboard Plug-in

To install the plug-in:

1. Start tcServer.
2. Connect through SAS Management Console to your middle tier. Click the **Folders** tab.
3. Navigate to **`/System/Applications/SAS BI Dashboard/BI Dashboard 4.4/`**:

- Import **$DFESP_HOME/etc/bid/esp.csx** into the **ContributorDefinitions/** folder.
- Import **$DFESP_HOME/etc/bid/esp.dsx** into the **DataSourceDefinitions/** folder.

4. Shutdown tcServer.
5. Copy the contents of **$DFESP_HOME/etc/bid/sas.bi.dashboard.war/plugins** to the **sas.bi.dashboard.war** directory on your tcServer install (for example, to the **/sas94/config/Lev1/Web/WebAppServer/SASServer1_1/sas_webapps/sas.bidashboard.war** directory.
6. Copy the following files to the **sas.bi.dashboard.war** directory on your tcServer installation:
   - **$DFESP_HOME/bin/dfx-esp-bid-plugin.jar**
   - **$DFESP_HOME/bin/dfx-esp-api.jar**
7. Start tcServer.

After completing these steps, you can create a subscriber stream in SAS BI Dashboard. Create a new indicator data and select **Event Stream Processing**. Then enter a valid event stream processing URL, which conforms to the format **dfESP://*host*:*port*/*project*/*contquery*/*window*?snapshot= true | false**, and click **Get Data**.

Chapter 10

Using the Publish/Subscribe API

Overview to the Publish/Subscribe API

The SAS Event Stream Processing Engine provides publish/subscribe application programming interfaces (APIs) for C and for Java. Use the publish/subscribe API to do the following:

- publish event streams into a running event stream processor project source window
- subscribe to an event stream window, either from the same machine or from another machine on the network

The publish/subscribe API is available on all architectures supported by SAS Event Stream Processing Engine. It is also supported on Microsoft 32-bit Windows® (which is publish/subscribe for the client only).

The publish/subscribe API handles cross-platform usage. For example, you can subscribe from Microsoft Windows to event streams in an SAS Event Stream Processing Engine running on Solaris SPARC even though the byte order Endianness is different.

You can also subscribe to an event stream so that it can be continuously loaded into a database for persistence. In this case, you more likely would use an event stream processor database connector or adapter.

Connectors are in-process classes that publish and subscribe to and from event stream processing windows. For more information, see "Using Connectors" on page 129.

Adapters are stand-alone executable files that publish and subscribe, potentially over a network. For more information, see Chapter 12, "Using Adapters," on page 161.

Note: The publish/subscribe API provides cross-platform connectivity and Endianness compatibility between the SAS Event Stream Processing Engine application and other networked applications, clients, and data feeds. The SAS Event Stream Processing Engine publish/subscribe API is IPv4 compliant.

Understanding Publish/Subscribe API Versioning

Publish/subscribe API versioning enables the server side of the client connection request to check the following information about the clients:

- protocol version
- command version (which is the release number)

It checks this information to determine whether it matches the server or is forward compatible with the server. Versioning was added to enable the SAS Event Stream Processing Engine to support forward compatibility for older publish/subscribe clients whenever feasible. When the server is initialized, the version number is logged using the following message:

```
dfESPengine version %s completed initialization
```

When a publish/subscribe client successfully connects, the following message is logged:

```
Client negotiation successful, client version: %d, server version: %d,
continuous query: %s, window: %s, total active clients = %d
```

On the other hand, when the client connection is incompatible, the following message is logged:

```
version mismatch; server is %d, client is %d
```

When the client version is unknown during the connection request (that is, a release earlier than 1.2), then the following message is logged:

```
Client version %d is unknown, and can be incompatible
```

You can read this log to determine the version number for the server and client. However, the success messages (like the server message from server initialize) are written using level information. Therefore, you see these only if you are logging messages (including informational and higher).

Using the C Publish/Subscribe API

The C Publish/Subscribe API from the Engine's Perspective

To enable publish/subscribe for the engine instance using the C++ Modeling API, you must provide a port number to the **`pubsub_ENABLE()`** parameter in the **`dfESPengine::initialize()`** call as follows:

```
dfESPengine *engine;
engine = dfESPengine::initialize(argc, argv, "engine",
pubsub_ENABLE(33335));

if (engine == NULL) {
     cerr <<"Error: dfESPengine::initialize() failed\n";
```

```
        return 1;
    }
```

Clients can use that port number (in this example 33335) to establish publish/subscribe connections. If publish/subscribe is not required, then use **pubsub_DISABLE** for that parameter.

To initialize publish/subscribe capabilities for a project, project->setPubSub() is called before calling engine->startProjects(). For example:

```
project->setPubSub(dfESPproject::ps_AUTO);
engine->startProjects();
```

This code opens a server listener socket on port 33335 to enable client subscribers and publishers to connect to the SAS Event Stream Processing Engine application or server for publish/subscribe services. After the connection request is made for publish/subscribe by a client (as described below), an ephemeral port is returned, which the publish/subscribe API uses for this connection. In cases when you need to override ephemeral ports for a specific port (for security purposes), specify **project->setPubSub** with a second parameter that is the preferred port to be used for the actual connections to this project. For example:

```
project->setPubSub(dfESPproject::ps_AUTO, 33444);
```

The first parameter of **project->setPubSub()** applies only to subscription services and it specifies how windows in the project are enabled to support client subscriptions. Specifying **ps_AUTO** enables clients to subscribe to all window output event streams in the project.

Alternatively, you can enable windows manually by specifying **ps_MANUAL**. For non-trivial projects, enable the specific windows of interest manually because automatically enabling all windows has a noticeable impact on overall performance. You can also specify **ps_NONE**, which disables subscribing for all windows.

If you use **ps_MANUAL** in **project->setPubSub()** to specify manual enablement of window subscribes, then use **enableWindowSubs()** for each desired window to enable the subscribe as follows:

```
project->enableWindowSubs(dfESPwindow *w);
```

If, however, you specified **ps_AUTO** or **ps_NONE** in **setPubSub()**, then subsequent calls to **enableWindowSubs()** are ignored and generate a warning.

Note: Clients can publish an event stream into any source window (and only source windows) in a project that is currently running. All source windows are enabled for publishing by default.

The C Publish/Subscribe API from the Client's Perspective

Clients that want to subscribe from or publish to the SAS Event Stream Processing Engine window event streams using the C API need to first initialize services on the client (using **C_dfESPpubsubInit()**). Next, start a subscription using **C_dfESPsubscriberStart()** and publisher using **C_dfESPpublisherStart()**, and then connect to the SAS Event Stream Processing Engine application or server using **C_dfESPpubsubConnect()**.

Clients that implement a publisher can then call **C_dfESPpublisherInject()** as needed to publish event blocks into the SAS Event Stream Processing Engine source window specified in the URL passed to **C_dfESPpublisherStart()**.

The specifics of the client publish/subscribe API are as follows.

Your client application must include the header file C_dfESPpubsubApi.h to provide publisher and subscriber services. In addition to the API calls, this file also defines the signatures of the user-supplied callback functions, of which there are currently two: the subscribed event block handler and the publish/subscribe failure handler.

The subscribed event block handler is used only by subscriber clients. It is called when a new event block from the SAS Event Stream Processing Engine application or server arrives. After processing the event block, the client is responsible for freeing it by calling **C_dfESPeventblock_destroy()**. The signature of this user-defined callback is as follows, where "**eb**" is the event block just read, "**schema**" is the schema of the event for client processing, and **ctx** is an optional context object containing call state:

```
typedef void (*C_dfESPsubscriberCB_func)(C_dfESPeventblock eb,
     C_dfESPschema schema, void *ctx);
```

The second callback function, **C_dfESPpubsubErrorCB_func()**, is optional for both subscriber and publisher clients. If supplied (that is, no NULL), it is called for every occurrence of an abnormal event within the client services, such as an unsolicited disconnect. This enables the client to handle and possibly recover from publish/subscribe services errors. The signature for this callback function is below, where the following is true:

- **failure** is either **pubsubFail_APIFAIL**, **pubsubFail_THREADFAIL**, or **pubsubFail_SERVERDISCONNECT**
- **code** provides the specific code of the failure
- **ctx** is an optional context object containing call state

```
typedef void (*C_dfESPpubsubErrorCB_func)(C_dfESPpubsubFailures
     failure, C_dfESPpubsubFailureCodes code);
```

The **C_dfESPpubsubFailures** and **C_dfESPpubsubFailureCodes** enums are defined in **C_dfESPpubsubFailures.h**.

A publisher client uses the **C_dfESPpublisherInject()** API function to publish event blocks into a source window in the SAS Event Stream Processing Engine application or server. The event block is injected into the source window running in the continuous query and project specified in the URL passed to **C_dfESPpublisherStart()**. A client can publish events to multiple windows in a project by calling **C_dfESPpublisherStart()** once for each window and then passing the appropriate client object to **C_dfESPpublisherInject()** as needed.

Finally, a client can query the SAS Event Stream Processing Engine application or server at any time to discover currently running windows, continuous queries, and projects in various granularities. This information is returned to the client in the form of a list of strings that represent names, which might subsequently be used to build URL strings to pass to **C_dfESPsubscriberStart()** or **C_dfESPpublisherStart()**. See the function description for a list of supported queries.

Functions for the C Publish/Subscribe API

The functions provided for client publish/subscribe in the publish/subscribe API are as follows. You can use them for simple connections or for more robust and complex connections with multiple connections or recovery handling by the client.

int C_dfESPpubsubInit(C_dfESPLoggingLevel *level*, const char **logConfigPath*)

Parameters:

level
the logging level

logConfigPath
the full pathname to the log configuration file

Return values:

1
success

0
failure — an error is logged to the SAS Event Stream Processing Engine log

Note: This function initializes SAS Event Stream Processing Engine client publisher and subscriber services, and must be called (only once) before making any other client calls, with the exception of **`C_dfESPpubsubSetPubsubLib()`**.

clientObjPtr C_dfESPpublisherStart(char **serverURL*, C_dfESPpubsubErrorCB_func *errorCallbackFunction*, void **ctx*)

Parameters:

serverURL
string representing the destination host, port, project, continuous query, and window

serverURL format
"**`dfESP://host:port/project/contquery/window`**"

errorCallbackFunction
either NULL or a user-defined function pointer for handling client service failures

`ctx`
optional context pointer for passing state into this call

Return value: a pointer to a client object that is passed to all API functions described below or NULL if there was a failure (error logged to the SAS Event Stream Processing Engine log).

Note: This function validates and retains the connection parameters for a specific publisher client connection.

clientObjPtr C_dfESPGDpublisherStart()

Parameters: Same parameters and return value as **`C_dfESPpublisherStart()`**. Additional required parameter: an acknowledged or not acknowledged callback function pointer. Additional required parameter: filename of this publisher's guaranteed delivery configuration file.

clientObjPtr C_dfESPsubscriberStart(char *_serverURL_, C_dfESPsubscriberCB_func _callbackFunction_, C_dfESPpubsubErrorCB_ func _errorCallbackFunction_, void *_ctx_)

Parameters:

serverURL
string representing the destination host, port, project, continuous query, and window in the SAS Event Stream Processing Engine. Also specifies the client snapshot requirement - if "true" the client receives the current snapshot state of the window prior to any incremental updates.

serverURL format
"**dfESP://*host*:*port*/*project*/*contquery*/*window*?snapshot=true |false**"

callbackFunction
a pointer to a user-defined function for handling received event blocks. This function must call **C_dfESPeventblock_destroy()** to free the event block.

errorCallbackFunction
either NULL or a user-defined function pointer for handling subscription service failures

ctx
optional context pointer for parsing state into this call

Return value: a pointer to a client object that is passed to all API functions described below, or NULL if there was a failure (error logged to the SAS Event Stream Processing Engine log).

Note: This function validates and retains the connection parameters for a specific subscriber client connection.

clientObjPtr C_dfESPGDsubscriberStart()

Parameters: Same parameters and return value as **C_dfESPsubscriberStart()**. Additional required parameter: filename of this subscriber's guaranteed delivery configuration file.

int C_dfESPpubsubConnect(clientObjPtr _client_)

Parameter:

client
pointer to a client object returned by
C_dfESPsubscriberStart() or
C_dfESPpublisherStart() or
C_dfESPGDsubscriberStart() or
C_dfESPGDpublisherStart()

Return values:

1
success

0
failure — error logged to the SAS Event Stream Processing Engine log

Note: This function attempts to establish a connection with the SAS Event Stream Processing Engine application or server.

int C_dfESPpubsubDisconnect(clientObjPtr *client*, int *block*)

Parameters:

client
pointer to a client object returned by
C_dfESPsubscriberStart() or
C_dfESPpublisherStart() or
C_dfESPGDsubscriberStart() or
C_dfESPGDpublisherStart()

block
set to 1 to wait for all queued events to be processed, else 0

Return values:

1
success

0
failure — error logged to the SAS Event Stream Processing Engine log

Note: This function closes the connection associated with the passed client object.

int C_dfESPpubsubStop(clientObjPtr *client*, int *block*)

Parameters:

client
pointer to a client object returned by
C_dfESPsubscriberStart() or
C_dfESPpublisherStart() or
C_dfESPGDsubscriberStart() or
C_dfESPGDpublisherStart()

block
set to 1 to wait for all queued events to be processed, else 0

Return values:

1
success

0
failure — error logged to the SAS Event Stream Processing Engine log

Note: This function stops the client session and removes the passed client object.

int C_dfESPpublisherInject(clientObjPtr *client*, C_dfESPeventblock *eventBlock*)

Parameters:

client
pointer to a client object returned by
C_dfESPpublisherStart()or
C_dfESPGDsubscriberStart()

eventBlock
the event block to inject into the SAS Event Stream Processing Engine. The block is injected into the source window, continuous query, and project associated with the passed client object.

Return values:

1
success

0
failure — error logged to the SAS Event Stream Processing Engine log

int C_dfESPpublisherInject(clientObjPtr *client*, C_dfESPeventblock *eventBlock*)

Note: This function implements the client publisher function by publishing events into the SAS Event Stream Processing Engine. Event blocks can be built using other additional functions provided in the event stream processor objects C API.

C_dfESPstringV C_dfESPpubsubQueryMeta(char **queryURL*)

Parameter: ***queryURL***
string representing the query to be posted to the SAS Event Stream Processing Engine.

Return value: a vector of strings representing the list of names comprising the response to the query, or NULL if there was a failure (error logged to the SAS Event Stream Processing Engine log). The end of the list is denoted by an empty string. The caller is responsible for freeing the vector by calling **`C_dfESPstringV_free()`**.

Note: This function implements a general event stream processor metadata query mechanism to allow a client to discover projects, continuous queries, windows, window schema, and window edges currently running in the SAS Event Stream Processing Engine. It has no dependencies or interaction with any other activity performed by the client. It opens an independent socket to send the query and closes the socket upon receiving the query reply.

Supported formats of *queryURL*

`"dfESP://host:port?get=projects"`	returns names of currently running projects
`"dfESP://host:port?get=projects_pubsubonly"`	returns names of currently running projects with publish/subscribe enabled
`"dfESP://host:port?get=queries"`	returns names of continuous queries in currently running projects
`"dfESP://host:port?get=queries_pubsubonly"`	returns names of continuous queries containing publish/subscribe enabled windows in currently running projects
`"dfESP://host:port?get=windows"`	returns names of windows in currently running projects

Supported formats of *queryURL*	
"dfESP://*host*:*port*?get=windows_pubsubonly"	returns names of publish/subscribe enabled windows in currently running projects
"dfESP://*host*:*port*/*project*?get=windows"	returns names of windows in the specified project, if running
"dfESP://*host*:*port*/*project*? get=windows_pubsubonly"	returns names of publish/subscribe-enabled windows in the specified project, if running
"dfESP://*host*:*port*/*project*?get=queries"	returns names of continuous queries in the specified project, if running
"dfESP://*host*:*port*/*project*? get=queries_pubsubonly"	returns names of continuous queries containing publish/ subscribe-enabled windows in the specified project, if running
dfESP://*host*:*port*/*project*/*contquery*? get=windows"	returns names of windows in the specified continuous query and project, if running
dfESP://*host*:*port*/*project*/*contquery*? get=windows_pubsubonly"	returns names of publish/subscribe-enabled windows in the specified continuous query and project, if running
dfESP://*host*:*port*/*project*/*contquery*/*window*? get=schema"	returns a single string that is the serialized version of the window schema
dfESP://*host*:*port*/*project*/*contquery*/*window*? get=edges"	returns the names of all the window's edges

C_dfESPstringV C_dfESPpubsubGetModel(char *_queryURL_)

Parameter:

queryURL
string representing the query to be posted to the SAS Event Stream Processing Engine.

Supported formats of *queryURL* are as follows:

- "**dfESP://*host*:*port***" – returns names of all windows in the model and their edges
- "**dfESP://*host*:*port*/*project***" – returns names of all windows in the project and their edges
- "**dfESP://*host*:*port*/*project*/*contquery***" – returns names of all windows in the continuous query and their edges

Return value: A vector of strings representing the response to the query, or NULL if there was a failure (error logged to the Event Stream Processing Engine log). The format of each string is "*project*/*query*/*window*: *edge1*, *edge2*, ...". The end of the list is denoted by an empty string. The caller is responsible for freeing the vector by calling **C_dfESPstringV_free()**.

Note: This function allows a client to discover a SAS Event Stream Processing Engine model by returning the complete set of windows in the model or project or continuous query, along with the window's edges. It has no dependencies or interaction with any other activity performed by the client. It opens an independent socket to send the query and closes the socket upon receiving the query reply.

void C_dfESPpubsubShutdown()

Shutdown publish/subscribe services

int C_dfESPpubsubPersistModel(char *_hostportURL_, const char *_persistPath_)

Parameters:

hostportURL
string in the form "**dfESP://*host*:*port***"

persistpath
the absolute or relative pathname for the persist file on the target platform

Return values:

1
success

0
failure — error logged to the SAS Event Stream Processing Engine log

Note: This function instructs the engine at the *hostportURL* to persist its current state to disk. It has no dependencies or interaction with any other activity performed by the client. It opens an independent socket to send the request and closes the socket upon receiving the request return code.

int C_dfESPpubsubQuiesceProject(char ****projectURL*, clientObjPtr *client*)**

Parameters:

projectURL
string in the form **"dfESP://*host*:*port*/*project*"**

client
optional pointer to a client object returned by **C_dfESPsubscriberStart()** or **C_dfESPpublisherStart()** or **C_dfESPGDsubscriberStart()** or **C_dfESPGDpublisherStart()**. If not null, wait for the specified client's queued events to be processed before quiescing the project.

Return values:

1
success

0
failure — error logged to the SAS Event Stream Processing Engine log.

Note: This function instructs the engine at the ***projectURL*** to quiesce the project in ***projectURL***. This call is synchronous, meaning that when it returns the project has been quiesced.

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Parameters:

psLib
Number representing the client/server transport

Supported values of *psLib* are as follows:

- ESP_PSLIB_NATIVE (default)
- ESP_PSLIB_SOLACE — In this mode a client configuration file named solace.cfg must be present in the current directory to provide appliance connectivity parameters.

For C and C++ clients, the text file format is as follows:

```
solace
{
SESSION_HOST = "10.37.150.244:55555"
SESSION_USERNAME = "pub1"
SESSION_PASSWORD = "pub1"
SESSION_VPN_NAME = "SAS"
SESSION_RECONNECT_RETRIES = "3"
SESSION_REAPPLY_SUBSCRIPTIONS = true
SESSION_TOPIC_DISPATCH = true
}
sas
{
buspersistence = false
queuename = "myqueue"
protobuf = false
protofile = "./GpbHistSimFactory.proto"
protomsg = "GbpTrade"
}
```

For Java clients, the text file format is as follows:

```
{
    solace =
     {
         session = ( "host", "10.37.150.244:55555",
                     "username", "sub1", "password",
                     "sub1", "vpn_name", "SAS");
         context = ( "CONTEXT_TIME_RES_MS", "50",
                     "CONTEXT_CREATE_THREAD", "1" );
     }
   sas=
    {
      buspersistence = false;
      queuename = "myqueue";
    protobuf = false;
    protofile = "./GpbHistSimFactory.proto";
    protomsg = "GpbTrade";
     }
 }
```

- ESP_PSLIB_TERVELA — In this mode a client configuration file named client.config must be present in the current directory to provide appliance connectivity parameters.

The text file format is as follows:

```
USERNAME            esp
PASSWORD            esp
PRIMARY_TMX         10.37.8.175
LOGIN_TIMEOUT       45000
GD_CONTEXT_NAME     tvaIF
GD_MAX_OUT          10000
```

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Return values:

1
success

0
failure

Note: This function call is optional, but if called it must be called before calling **`C_dfESPpubsubInit()`**. It modifies the transport used between the client and the SAS Event Stream Processing engine from the default peer-to-peer TCP/IP based socket connection that uses the ESP publish/subscribe protocol. Instead, you can specify ESP_PSLIB_SOLACE or ESP_PSLIB_TERVELA to indicate that the client's TCP/IP peer is a Solace or Tervela appliance, respectively. This mode requires that the SAS Event Stream Processing engine runs a Solace or Tervela connector to provide the corresponding inverse client to the appliance. The topic names used by the appliance are coordinated by the publish/subscribe client and connector to correctly route event blocks through the appliance.

Note: Solace functionality is not available on HP Itanium, AIX, and 32–bit Microsoft Windows platforms.

Note: Tervela functionality is not available on HP Itanium, AIX, SPARC, and 32–bit Microsoft Windows platforms.

Note: When using the Solace or Tervela transports, the following publish/subscribe API functions are not supported:

```
C_dfESPpubsubGetModel()
C_dfESPGDpublisherStart()
C_dfESPGDpublisherGetID()
C_dfESPGDsubscriberStart()
C_dfESPGDsubscriberAck()
C_dfESPpubsubSetBufferSize()
```

C_dfESPGDsubscriberAck()

Parameters: Triggers an acknowledged.

- Parameter 1: client object pointer.
- Parameter 2: event block pointer. The event block must not be freed before this function returns.

Return values:

- 1 = success
- 0 = failure

C_dfESPGDpublisherCB_func()

Parameters: Signature of the new publisher callback function passed to **`C_dfESPGDpublisherStart()`**

- Parameter 1: READY or ACK or NACK (acknowledged or not acknowledged).
- Parameter 2: 64-bit event block ID
- Parameter 3: the user context pointer passed to **`C_dfESPGDpublisherStart()`**

C_dfESPGDpublisherCB_func()

Return value: Void

C_dfESPGDpublisherGetID()

Return value: 64-bit event block ID. Might be called by a publisher to obtain sequentially unique IDs to be written to event blocks before injecting them to a guaranteed delivery-enabled publish client.

int C_dfESPpubsubSetBufferSize(clientObjPtr *client*, int32_t *mbytes*)

Parameters:

client
pointer to a client object returned by **C_dfESPsubscriberStart()** ,**C_dfESPpublisherStart()** , **C_dfESPGDsubscriberStart()**, or **C_dfESPGDpublisherStart()**

mbytes
the read and write buffer size, in units of 1MB

Return values:

1
success

0
failure

Note: This function call is optional, but if called it must be called after **C_dfESPsubscriberStart()**, **C_dfESPpublisherStart()**, **C_dfESPGDsubscriberStart()**, or **C_dfESPGDpublisherStart()** and before **C_dfESPpubsubConnect()**. It modifies the size of the buffers used for socket Read and Write operations. By default this size is 16MB

protobuffObjPtr C_dfESPpubsubInitProtobuff(char **protoFile*, char **msgName*, C_dfESPschema *C_schema*)

Parameters:

protoFile
Path to the Google **.proto** file that contains the message definition.

msgName
Name of the target message definition in the Google **.proto** file.

C_schema
Pointer to the window schema.

Return value: A pointer to a **protobuff** object, which is passed to all other **protobuff** API functions. NULL when there is a failure.

C_dfESPeventblock C_dfESPprotobuffToEb(protobuffObjPtr *protobuff*, void **serializedProtobuff*)

Parameters:

protobuff
pointer to the object created by `C_dfESPpubsubInitProtobuff()`

serializedProtobuff
pointer to serialized `protobuff` received on a socket

Return value: Event block pointer

void *C_dfESPebToProtobuff(protobuffObjPtr *protobuff*, C_dfESPeventblock *C_eb*, int32_t *index*)

Parameters:

protobuff
pointer to the object created by `C_dfESPpubsubInitProtobuff()`

C_eb
event block pointer

index
index of the event to convert in the event block

Return value: pointer to serialized `protobuff`

void C_dfESPdestroyProtobuff(protobuffObjPtr *protobuff*, void **serializedProtobuff*)

Parameters:

protobuff
pointer to the object created by `C_dfESPpubsubInitProtobuff()`

serializedProtobuff
pointer to serialized `protobuff` received on a socket

int C_dfESPsubscriberMaxQueueSize(char **serverURL*, int *maxSize*, int *block*)

serverURL a string of one of the following forms:

`"dfESP://host:port"`

`"dfESP://host:port/project"`

`"dfESP://host:port/project/contquery"`

`"dfESP://host:port/project/contquery/window"`

maxSize the maximum number of event blocks that can be queued for any single subscriber to a window in the *serverURL*

block set to 1 to wait for queue size to fall below *maxSize*, else disconnect the client

int C_dfESPsubscriberMaxQueueSize(char *serverURL, int maxSize, int block)	
Return values	1 — success 0 — failure — error logged to the SAS Event Stream Processing Engine log

A C library provides a set of functions to enable SAS Event Stream Processing Engine client developers to analyze and manipulate the event stream processing objects from the SAS Event Stream Processing Engine application or server. These functions are a set of C wrappers around a small subset of the methods provided in the C++ Modeling API. With these wrappers, client developers can use C rather than C++. Examples of these objects are events, event blocks, and schemas. A small sampling of these calls follows. For the full set of calls, see the API reference documentation available at **$DFESP_HOME/doc/html**.

To get the size of an event block:	`C_ESP_int32_t eventCnt = C_dfESPeventblock_getSize(eb);`
To extract an event from an event block:	`C_dfESPevent ev = C_dfESPeventblock_getEvent(eb, eventIndx);`
To create an object (a string representation of schema in this case):	`C_ESP_utf8str_t schemaCSV = C_dfESPschema_serialize(schema);`
To free an object (a vector of strings in this case):	`C_dfESPstringV_free(metaVector);`

Using the Java Publish/Subscribe API

Overview to the Java Publish/Subscribe API

The SAS Event Stream Processing Engine and its C publish/subscribe API use the SAS logging library, whereas the Java publish/subscribe API uses the Java logging APIs in the **java.util.logging** package. Please refer to that package for log levels and specifics about Java logging.

The Java publish/subscribe API is provided in two packages. These packages define the following public interfaces:

- **sas.com.dfESP.api.pubsub**
  - **sas.com.dfESP.api.pubsub.clientHandler**
  - **sas.com.dfESP.api.pubsub.clientCallbacks**
- **sas.com.dfESP.api.server**

- **sas.com.dfESP.api.server.datavar**
- **sas.com.dfESP.api.server.event**
- **sas.com.dfESP.api.server.eventblock**
- **sas.com.dfESP.api.server.library**
- **sas.com.dfESP.api.server.schema**

A client can query the Event Stream Processor application or server at any time to discover currently running windows, continuous queries, and projects in various granularities. This information is returned to the client in the form of a list of strings that represent names. This list can be used to build URL strings to pass to **subscriberStart()** or **publisherStart()**.

The Java publish/subscribe API provides the same ability as the C publish/subscribe API to substitute a Solace or Tervela transport, but it is invoked in a different way. Instead of calling an API function to modify the transport, simply specify **dfx-esp-tervela-api.jar** or **dfx-esp-solace-api.jar** in the class path, in front of **dfx-esp-api.jar**.

You can find details about these interface methods and their parameters in several places:

- The parameters and usage for the Java publish/subscribe API are the same for the equivalent calls for the C publish/subscribe API.
- The API references available at **$DFESP_HOME/doc/html**.
- The interface references available at **$DFESP_HOME/doc/html**.
- The reference implementations for each of the interface references.

Using High Level Publish/Subscribe Methods

The following high-level publish/subscribe methods are defined in the following interface reference: **sas.com.dfESP.api.pubsub.clientHandler**.

Method	Description
boolean init(Level *level*)	Initialize publish/subscribe services
dfESPclient publisherStart(String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*)	Start a publisher.
dfESPclient subscriberStart(String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*)	Start a subscriber
boolean connect(dfESPclient *client*)	Connect to the Event Stream Processor application or server
boolean publisherInject((dfESPclient *client*, dfESPeventblock eventblock)	Publish event blocks
ArrayList< String > *queryMeta* (String *queryURL*)	Query model metadata

Method	Description
ArrayList< String > getModel(String *queryURL*)	Query model windows and their edges
boolean disconnect (dfESPclient *client*, boolean *block*)	Disconnect from the event stream processor
boolean stop (dfESPclient *client*, boolean *block*)	Stop a subscriber or publisher
void shutdown ()	Shutdown publish/subscribe services
boolean setBufferSize(dfESPclient *client*, int *mbytes*)	Change the default socket read and write buffer size
dfESPclient GDsubscriberStart (String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*, String *configFile*)	Start a guaranteed delivery subscriber
dfESPclient GDpublisherStart (String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*, String *configFile*)	Start a guaranteed delivery publisher
long GDpublisherGetID()	Get a sequentially unique ID to write to an event block to be published using guaranteed delivery
boolean GDsubscriberAck(dfESPclient *client*, dfESPeventblock *eventblock*)	Trigger a guaranteed delivery acknowledgment
boolean persistModel(String *hostportURL*, String *persistPath*)	Instruct a running engine to persist its current state to disk
boolean quiesceProject(String *projectURL*, dfESPclient *client*)	Instruct a running engine to quiesce a specific project in the model.
boolean subscriberMaxQueueSize(String *serverURL*, int *maxSize*, boolean *block*)	Configure the maximum size of the queues in the event stream processing server that are used to enqueue event blocks sent to subscribers. Set **block** to 1 to wait for queue size to fall below**maxSize**, else disconnect the client.

For more information, see**$DFESP_HOME/doc/html/index.html**. Search the **Classes** page for **clientHandler**.

Using Methods That Support Google Protocol Buffers

The following methods support Google Protocol Buffers. They are defined in this interface reference: **sas.com.dfESP.api.pubsub.protobufInterface**.

Method	Description
boolean init(String *fileDescriptorSet*, String *msgName*, dfESPschema *schema*)	Initialize the library that supports Google Protocol Buffers.
dfESPeventblock protobufToEb(byte[] *serializedProtobuf*)	Convert a **protobuf** message to an event block.
byte[] ebToProtobuf(dfESPeventblock *eb*, int *index*)	Convert an event in an event block to a **protobuf** message.

For more information, see "Publish/Subscribe API Support for Google Protocol Buffers" on page 227.

Using User-supplied Callback Functions

The **sas.com.dfESP.api.pubsub.clientCallbacks** interface reference defines the signatures of the user-supplied callback functions. There currently are three functions:

- the subscribed event block handler
- the publish/subscribe failure handler
- the guaranteed delivery ACK-NACK handler

The subscribed event block handler is used only by subscriber clients. It is called when a new event block from the application or server arrives. After processing the event block, the client is responsible for freeing it by calling **eventblock_destroy()**. The signature of this user-defined callback is as follows where "**eventBlock**" is the event block just read, "**schema**" is the schema of the event for client processing, and "**ctx**" is an optional context pointer for maintaining call state:

```
void sas.com.dfESP.api.pubsub.clientCallbacks.dfESPsubscriberCB_func
     (dfESPeventblock eventBlock, dfESPschema schema, Object ctx)
```

The second callback function for publish/subscribe client error handling is optional for both subscriber and publisher clients. If supplied (that is, not NULL), it is called for every occurrence of an abnormal event within the client services, such as an unsolicited disconnect. This enables the client to gracefully handle and possibly recover from publish/subscribe services errors. The signature for this callback function is below where

- **failure** is either **pubsubFail_APIFAIL**, **pubsubFail_THREADFAIL**, or **pubsubFail_SERVERDISCONNECT**.
- **code** provides the specific code of the failure.
- **ctx** is an optional context pointer to a state data structure.

```
void sas.com.dfESP.api.pubsub.clientCallbacks.dfESPpubsubErrorCB_func
```

```
        (clientFailures failure, clientFailureCodes code, Object ctx)
```

clientFailures and client FailureCodes are defined in interface references **sas.com.dfESP.api.pubsub.clientFailures** and **sas.com.dfESP.pubsub.clientFailureCodes**.

The guaranteed delivery ACK-NACK handler is invoked to provide the status of a specific event block, or to notify the publisher that all subscribers are connected and publishing can begin. The signature for this callback function is as follows:

```
void sas.com.dfESP.api.pubsub.clientCallbacks.dfESPGDpublisherCB_func
   (clientGDStatus eventBlockStatus, long eventBlockID, Object ctx)
```

where

- **eventBlockStatus** is either ESP_GD_READY, ESP_GD_ACK, or ESP_GD_NACK
- **eventBlockID** is the ID written to the event block prior to publishing
- **ctx** is an optional context pointer to a state data structure

Chapter 11

Using Connectors

Overview to Using Connectors

What Do Connectors Do?

Connectors use the publish/subscribe API to do one of the following:

- publish event streams into source windows. Publish operations do the following:

  - read event data (usually continuously) from the specified source
  - inject that event data (usually continuously) into a specific source window of a running event stream processor
- subscribe to window event streams. Subscribe operations write output events from a window of a running event stream processor to the specified target (usually in a continuous manner).

Connectors do not simultaneously publish and subscribe. You can find connectors in libraries located at **$DFESP_HOME/lib/plugins**. On Microsoft Windows platforms, you can find them at **%DFESP_HOME%\bin\plugins**.

All connector classes are derived from a base connector class that is included in a connector library. The base connector library includes a connector manager that is responsible for loading connectors during initialization. This library is located in **$DFESP_HOME/lib/libdfxesp_connectors-Maj.Min**, where **Maj.Min** indicates the release number for the distribution.

Connector Examples

Connector examples are available in **$DFESP_HOME/src**. The **sample_connector** directory includes source code for a user-defined connector derived from the **dfESPconnector** base class. It also includes sample code that invokes a sample connector. For more information about how to write a connector and getting it loaded by the connector manager, see "Writing a Custom Connector" on page 157.

The remaining connector examples implement application code that invokes existing connectors. These connectors are loaded by the connector manager at initialization. Those examples are as follows:

- db_connector_publisher
- db_connector_subscriber
- json_connector_publisher
- json_connector_subscriber
- socket_connector_publisher
- socket_connector_subscriber
- xml_connector_publisher
- xml_connector_subscriber

Obtaining Connectors

To obtain a new instance of a connector, call the **dfESPwindow::getConnector()** method. Pass the connector name as the first parameter:

```
dfESPConnector *subConn =

static_cast<dfESPCConnector *>(window->getConnector("sampleConnector", true));
```

The packaged connector names are as follows:

- "fs"
- "db"
- "smtp"

- “mq”
- “tva”
- “sol”
- “tdata”
- “tibrv”
- “pi”

Pass an **autoStart** Boolean as the second parameter. If this Boolean is set to false, the connector can be started later by calling the **dfESPwindow::startConnectors()** method or the **dfESPconnector::start()** method.

After a connector instance is obtained, any of its base class public methods can be called. This includes **setParameter()**, which can be called multiple times to set required and optional parameters. Parameters must be set before the connector is started.

The **type** parameter is required and is common to all connectors. It must be set to **pub** or **sub**.

Additional connector configuration parameters are required depending on the connector type, and are described later in this section.

Connectors can be stopped at any time by either calling the **dfESPwindow::stopConnectors()** method or the **dfESPconnector::stop()** method.

Activating Optional Plug-ins

The **$DFESP_HOME/lib/plugins** directory contains the complete set of plug-in objects supported by SAS Event Stream Processing engine. Plug-ins that contain "_cpi" in their filename are connectors.

The Connectors.Excluded file in the **$DFESP_HOME/lib** directory contains a list of connectors. When the connector manager starts, SAS Event Stream Processing loads all connectors found in the **/plugins** directory, except those that are listed in connectors.excluded. By default, connectors.excluded specifies connectors that require third-party libraries that are not shipped with SAS Event Stream Processing Engine. This prevents those connectors from being automatically loaded and generating errors due to missing dependencies.

You can edit connectors.excluded as needed. The complete list of valid names for connectors.excluded is as follows:

- db
- fs
- smtp
- mq
- tibrv
- sol
- tdata
- tva
- pi

Using File and Socket Connectors

Overview to File and Socket Connectors

File and socket connectors support both publish and subscribe operations on files or socket connections that stream the following data types:

- **binary**
- **csv**
- **xml**
- **json**
- **syslog** (only supports publish operations)
- **sashdat** (only supports subscribe operations, and only as a client type socket connector)

The file or socket nature of the connector is specified by the form of the configured **fsname**. A name in the form of *host*: *port* is a socket connector. Otherwise, it is a file connector.

When the connector implements a socket connection, it might act as a client or server, regardless of whether it is a publisher or subscriber. When you specify both *host* and *port* in the **fsname**, the connector implements the client. The configured host and port specify the network peer implementing the server side of the connection. However, when *host* is blank (that is, when **fsname** is in the form of ": *port*"), the connection is reversed. The connector implements the server and the network peer is the client.

Use the following parameters when you specify file and socket connectors.

Table 11.1 *Required Parameters for File and Socket Connectors*

Parameter	Description
type	Specifies whether to publish or subscribe
fstype	**binary/csv/xml/json/syslog/hdat**
fsname	Specifies the input file for publishers, output file for subscribers, or socket connection in the form of *host*: *port* . Leave *host* blank to implement a server instead of a client.

Table 11.2 *Optional Parameters for Subscriber File and Socket Connectors*

Parameter	Description
snapshot	Disables the sending of snapshot data. The default value is enabled.
collapse	Converts UPDATE_BLOCK events to UPDATE events in order to make subscriber output publishable. The default value is disabled.

Parameter	Description
periodicity	Specifies the interval in seconds at which the subscriber output file is closed and a new output file opened. When configured, a timestamp is appended to all output filenames for when the file was opened. This parameter does not apply to socket connectors with **fstype** other than **hdat**.
maxfilesize	Specifies the maximum size in bytes of the subscriber output file. When reached, a new output file is opened. When configured, a timestamp is appended to all output filenames. This parameter does not apply to socket connectors with **fstype** other than **hdat**.
hdatfilename	Specifies the name of the Objective Analysis Package Data (HDAT) file to be written to the Hadoop Distributed File System (HDFS). Applies only to and is required for the **hdat** connector.
hdfsblocksize	Specifies in Mbytes the block size used to write a Objective Analysis Package Data (HDAT) file. Applies only to and is required for the **hdat** connector.
hdatmaxstringlength	Specifies in bytes the fixed size of string fields in Objective Analysis Package Data (HDAT) files. You must specify a multiple of 8. Strings are padded with spaces. Applies only to and is required for the **hdat** connector.
hdatnumthreads	Specifies the size of the thread pool used for multi-threaded writes to data node socket connectors. A value of 0 or 1 indicates that writes are single-threaded and use only a name-node connection. Applies only to and is required for the **hdat** connector.
hdatmaxdatanodes	Specifies the maximum number of data node connectors. The default value is the total number of live data nodes known by the name mode. This parameter is ignored when **hdatnumthreads** <=1. Applies only to the **hdat** connector.
hdfsnumreplicas	Specifies the number of Hadoop Distributed File System (HDFS) replicas created with writing a Objective Analysis Package Data (HDAT) file. The default value is 1. Applies only to the **hdat** connector.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.

Table 11.3 *Optional Parameters for Publisher File and Socket Connectors*

Parameter	Description
blocksize	Specifies the number of events to include in a published even block. The default value is 1.
transactional	Sets the event block type to transactional. The default value is normal.

Parameter	Description
growinginputfile	Enables reading from a growing input file by publishers. The default value is disabled. When enabled, the publisher reads indefinitely from the input file until the connector is stopped or the server drops the connection. This parameter does not apply to socket connectors.
rate	Controls the rate at which event blocks are injected. Specified in events per seconds.
maxevents	Specifies the maximum number of events to publish.
prebuffer	Controls whether event blocks are buffered to an event block vector before doing any injects. The default value is false. Not valid with **growinginputfile** or for a socket connector
header	Specifies the number of input lines to skip before starting publish operations. The default value is 0. This parameter only applies to csv and syslog publish connectors.

File and Socket Connector Publisher Blocking Rules

Input data used by a file and socket connector publisher can contain optional transaction delimiters. These transaction delimiters are supported only within input data of type XML or JSON. If present, a pair of these delimiters defines a block of contained events, and the publisher ignores any configured value for the **blocksize** parameter. If transaction delimiters are not present in input data, the event block size is equal to the **blocksize** parameter. If the **blocksize** parameter is not configured, the default block size is 1. The file and socket connector subscriber always includes transaction delimiters in the output file.

XML File and Socket Connector Data Format

The XML file and socket connector subscriber writes the following header to XML output:

```
<?xml version="1.0" encoding="utf-8"?>
```

The same header is support by the XML file and socket publisher. The following XML tags are valid:

- **<project>**
- **<contquery>**
- **<window>**
- **<transaction>**
- **<event>**
- **<opcode>**

In addition, any tags that correspond to event data field names are valid when contained within an event. All of these tags are required except for "transaction". For more information, see “File and Socket Connector Publisher Blocking Rules” on page 134.

Event data field tags can also contain a corresponding Boolean attribute named "key," which identifies a field as a key field. If not present, its default value is "false."

Key fields must match key fields in the window schema, and must be present with **value="true"** in all events in XML input data.

Valid data for the **opcode** tag in input data includes the following:

"opcode" Tag Value	Opcode
"**i** "	**Insert**
"**u**"	**Update**
"**p**"	**Upsert**
"**d**"	**Delete**
"**s**"	**Safedelete**

The subscriber writes only Insert, Update, and Delete opcodes to the output data.

The **opcode** tag can contain a **flags** attribute, where its only valid value is "**p**". This identifies the event as a partial update record.

Non-key fields in the event are not required in input data, and their **value = NULL** (or partial-update) if missing, or if the fields contain no data. The subscriber always writes all event data fields.

JSON File and Socket Connector Data Format

For allowed fields, the JSON format mirrors the XML tags format and hierarchy, except attributes in XML data (such as "key" and "flags"). These are represented in JSON as additional fields enclosed in the parent field.

All of the rules described in the XML data format apply to JSON as well.

In addition, JSON input data can contain events inside an array, if the complete array is contained within a single parent transaction or window. This is optional and is only a syntactical convenience. The subscriber does not write event arrays to the output data.

Syslog File and Socket Connector Notes

The syslog file and socket connector is supported only for publisher operations. It collects syslog events, converts them into ESP event blocks, and injects them into one or more source windows in a running ESP model.

The input syslog events are read from a file or named pipe written by the syslog daemon. Syslog events filtered out by the daemon are not seen by the connector. When reading from a named pipe, the following conditions must be met:

- The connector must be running in the same process space as the daemon.
- The pipe must exist.
- The daemon must be configured to write to the named pipe.

You can specify the **growinginputfile** parameter to read data from the file or named pipe as it is written.

The connector reads text data one line at a time, and parses the line into fields using spaces as delimiters.

The fields in the corresponding window schema must be of type ESP_UTF8STR or ESP_DATETIME. The number of schema fields must not exceed the number of fields parsed in the syslog file.

HDAT Subscribe Socket Connector Notes

This connector writes Objective Analysis Package Data (HDAT) files in SASHDAT format. This socket connector connects to the name node specified by the **fsname** parameter. If run multi-threaded (as specified by the **hdatnumthreads** parameter), the connector opens socket connections to all the data nodes that are returned by the name node. It then writes subscriber event data as Objective Analysis Package Data (HDAT) file parts to all data nodes using the block size specified by the **hdfsblocksize** parameter. These file parts are then concatenated to a single file when the name node is instructed to do so by the connector.

The connector automatically concatenates file parts when stopped. It might also stop periodically as specified by the optional **periodicity** and **maxfilesize** parameters.

Because SASHDAT format consists of static row data, the connector ignores Delete events. It treats Update events the same as Insert events.

Using the Database Connector

Overview to Using the Database Connector

The database connector provided by SAS Event Stream Processing Engine supports both publish and subscribe operations. It uses the DataDirect ODBC driver. Currently, it is certified for the following databases:

- Oracle
- MySQL
- IBM DB2
- Greenplum
- PostgreSQL
- SAP Sybase ASE
- Teradata
- Microsoft SQL Server
- IBM Informix
- Sybase IQ

The connector requires that database connectivity be available through a system Data Source Name (DSN). This DSN and the associated database user credentials are required configuration parameters for the connector.

For SAS Event Stream Processing Engine installations not on Microsoft Windows, the DataDirect drivers are located in **$DFESP_HOME/lib**. The DSN required for your specific database connection is configured in an odbc.ini file that is pointed to by the ODBC_INI environment variable. A default odbc.ini.template file is available in

$DFESP_HOME/etc. Alternatively, two useful tools to configure and verify the DataDirect drivers and your database connection are available in **$DFESP_HOME/bin**:

- **dfdbconf** - Use this interactive ODBC Configuration Tool to add an ODBC DSN. Run **$DFESP_HOME/bin/dfdbconf**. Select a driver from the list of available drivers and set the appropriate parameters for that driver. The new DSN is added to the odbc.ini file.
- **dfdbview** - This interactive tool enables the user to manually connect to the database and perform operations using SQL commands.

For Windows ESP installations, ensure that the optional ODBC component is installed during installation. Then you can configure a DSN using the Windows ODBC Data Source Administrator. This application is located in the **Windows Control Panel** under **Administrative Tools**. Beginning in Windows 8, the icon is named ODBC Data Sources. On 64-bit operating systems, there are 32-bit and 64-bit versions.

Perform the following steps to create a DSN in a Windows environment:

1. Enter **odbc** in the Windows search window. The default ODBC Data Source Administrator is displayed.
2. Select the **System DSN** tab and click **Add**.
3. Select the appropriate driver from the list and click **Finish**.
4. Enter your information in the Driver Setup dialog box.
5. Click **OK** when finished.

This DSN is supplied as a parameter to the database connector.

The connector publisher obtains result sets from the database using a single SQL statement configured by the user. Additional result sets can be obtained by stopping and restarting the connector. The connector subscriber writes window output events to the database table configured by the user.

Use the following parameters with database connectors

Table 11.4 *Required Parameters for Subscriber Database Connectors*

Parameter	Description
type	Specifies to subscribe.
connectstring	Specifies the database DSN and user credentials in the format "**DSN=** ***dsn*****;uid=*****userid*****;pwd=*****password*****;**"
tablename	Specifies the target table name.

Table 11.5 *Required Parameters for Publisher Database Connectors*

Parameter	Description
type	Specifies to publish.
connectstring	Specifies the database DSN and user credentials in the format "**DSN=*****dsn*****;uid=*****userid*****;pwd=*****password*****;**"

Parameter	Description
`selectstatement`	Specifies the SQL statement executed on the source database.

Table 11.6 *Optional Parameters for Subscriber Database Connectors*

Parameter	Description
`snapshot`	Disables the sending of snapshot data. The default value is enabled.
`rmretdel`	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.

Table 11.7 *Optional Parameters for Publisher Database Connectors*

Parameter	Description
`blocksize`	Specifies the number of events to include in a published event block. The default value is 1.
`transactional`	Sets the event block type to transactional. The default value is normal.

The number of columns in the source or target database table and their data types must be compatible with the schema of the involved event stream processor window.

Subscriber Event Stream Processor to SQL Data Type Mappings

For databases that have been certified to date, the event stream processor to SQL data type mappings are as follows.

Subscriber Event Stream Processor Data Type	SQL Data Type
ESP_UTF8STR	SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, SQL_WCHAR, SQL_WVARCHAR, SQL_WLONGVARCHAR, SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, SQL_GUID
ESP_INT32	SQL_INTEGER, SQL_BIGINT, SQL_DECIMAL, SQL_BIT, SQL_TINYINT, SQL_SMALLINT
ESP_INT64	SQL_BIGINT, SQL_DECIMAL, SQL_BIT, SQL_TINYINT, SQL_SMALLINT
ESP_DOUBLE	SQL_DOUBLE, SQL_FLOAT, SQL_REAL, SQL_NUMERIC, SQL_DECIMAL

Subscriber Event Stream Processor Data Type	SQL Data Type
ESP_MONEY	SQL_DOUBLE (converted to ESP_DOUBLE), SQL_FLOAT, SQL_REAL, SQL_NUMERIC (converted to SQL_NUMERIC), SQL_DECIMAL (converted to SQL_NUMERIC)
ESP_DATETIME	SQL_TYPE_DATE (only sets year/month/day), SQL_TYPE_TIME (only sets/hours/minutes/seconds), SQL_TYPE_TIMESTAMP (sets fractional seconds = 0)
ESP_TIMESTAMP	SQL_TYPE_TIMESTAMP

The following SQL mappings are not supported: SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, SQL_BIT, SQL_TINYINT, SQL_SMALLINT

Publisher SQL to Event Stream Processor Data Type Mappings

The SQL to event stream processor data type mappings are as follows:

Publisher SQL Data Type	Event Stream Processor Data Types
SQL_CHAR	ESP_UTF8STR
SQL_VARCHAR	ESP_UTF8STR
SQL_LONGVARCHAR	ESP_UTF8STR
SQL_WCHAR	ESP_UTF8STR
SQL_WVARCHAR	ESP_UTF8STR
SQL_WLONGVARCHAR	ESP_UTF8STR
SQL_BIT	ESP_INT32, ESP_INT64
SQL_TINYINT	ESP_INT32, ESP_INT64
SQL_SMALLINT	ESP_INT32, ESP_INT64
SQL_INTEGER	ESP_INT32, ESP_INT64
SQL_BIGINT	ESP_INT64
SQL_DOUBLE	ESP_DOUBLE, ESP_MONEY (upcast from ESP_DOUBLE)
SQL_FLOAT	ESP_DOUBLE, ESP_MONEY (upcast from ESP_DOUBLE)
SQL_REAL	ESP_DOUBLE

Publisher SQL Data Type	Event Stream Processor Data Types
SQL_TYPE_DATE	ESP_DATETIME (sets only year/month/day)
SQL_TYPE_TIME	ESP_DATETIME (sets only hours/minutes/seconds)
SQL_TYPE_TIMESTAMP	ESP_TIMESTAMP, ESP_DATETIME
SQL_DECIMAL	ESP_INT32 (only if scale = 0, and precision must be <= 10), ESP_INT64 (only if scale = 0, and precision must be <= 20), ESP_DOUBLE
SQL_NUMERIC	ESP_INT32 (only if scale = 0, and precision must be <= 10), ESP_INT64 (only if scale = 0, and precision must be <= 20), ESP_DOUBLE, ESP_MONEY (converted from SQL_NUMERIC)
SQL_BINARY	ESP_UTF8STR
SQL_VARBINARY	ESP_UTF8STR
SQL_LONGVARBINARY	ESP_UTF8STR

Using the SMTP Subscribe Connector

You can use the Simple Mail Transfer Protocol (SMTP) subscribe connector to e-mail window event blocks or single events, such as alerts or items of interest. This connector is subscribe-only. The connection to the SMTP server uses port 25. No user authentication is performed, and the protocol runs unencrypted.

The e-mail sender and receiver addresses are required information for the connector. The e-mail subject line contains a standard event stream processor URL in the form "**dfESP://*host*:*port*/*project*/*contquery*/*window***", followed by a list of the key fields in the event. The e-mail body contains data for one or more events encoded in CSV format.

The parameters for the SMTP connector are as follows:

Table 11.8 *Required Parameters for the SMTP Connector*

Parameter	Description
type	Specifies whether to publish or subscribe.
smtpserver	Specifies the SMTP server host name or IP address.
sourceaddress	Specifies the e-mail address to be used in the “from” field of the e-mail.
destaddress	Specifies the e-mail address to which to send the e-mail message.

Table 11.9 Optional Parameters for SMTP Connectors

Parameter	Description
snapshot	Disables the sending snapshot data. The default value is enabled.
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
emailperevent	Specifies true or false. The default is false. If false, each e-mail body contains a full event block. If true, each mail body contains a single event.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.

Using the IBM WebSphere MQ Connector

The IBM WebSphere MQ connector (MQ) supports the IBM WebSphere Message Queue Interface for publish and subscribe operations. The subscriber receives event blocks and publishes them to an MQ queue. The publisher is an MQ subscriber, which injects received event blocks into source windows.

The IBM WebSphere MQ Client run-time libraries must be installed on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (for example, specifying **LD_LIBRARY_PATH**on Linux platforms).

The connector operates as an MQ client. It requires that you define the environment variable **MQSERVER** to specify the connector's MQ connection parameters. This variable specifies the server's channel, transport type, and host name. For more information, see your WebSphere documentation.

The topic string used by an MQ connector is a required connector parameter. In addition, an MQ subscriber requires a parameter that defines the message format used to publish events to MQ. The format options are csv or binary. An MQ publisher can consume any message type that is produced by an MQ subscriber.

An MQ publisher requires two additional parameters that are related to durable subscriptions. The publisher always subscribes to an MQ topic using a durable subscription. This means that the publisher can re-establish a former subscription and receive messages that had been published to the related topic while the publisher was disconnected.

These parameters are as follows:

- subscription name, which is user supplied and uniquely identifies the subscription
- subscription queue, which is the MQ queue that is opened for input by the publisher

The MQ persistence setting of messages written to MQ by an MQ subscriber is always equal to the persistence setting of the MQ queue.

Use the following parameters for MQ connectors.

Table 11.10 *Required Parameters for Subscriber MQ Connectors*

Parameter	Description
type	Specifies to subscribe.
mqtopic	Specifies the MQ topic name.
mqtype	Specifies binary or CSV.

Table 11.11 *Required Parameters for Publisher MQ Connectors*

Parameter	Description
type	Specifies to publish.
mqtopic	Specifies the MQ topic name.
mqsubname	Specifies the MQ subscription name.
mqsubqueue	Specifies the MQ queue.

Table 11.12 *Optional Parameters for Subscriber MQ Connectors*

Parameter	Description
snapshot	Disables the sending of snapshot data. The default value is enabled.
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
queuemanager	Specifies the MQ queue manager.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.

Table 11.13 *Optional Parameters for Publisher MQ Connectors*

Parameter	Description
blocksize	Specifies the number of events to include in a published event block. The default value is 1.
transactional	Sets the event block type to transactional. The default value is normal.
queuemanager	Specifies the MQ queue manager.

Using the Tervela Data Fabric Connector

The Tervela Data Fabric connector communicates with a software or hardware-based Tervela Data Fabric for publish and subscribe operations.

A Tervela subscriber connector receives events blocks and publishes to the following Tervela topic:
"SAS.ENGINES.*enginename*.*projectname*.*queryname*.*windowname*.OUT."

A Tervela publisher connector reads event blocks from the following Tervela topic: "SAS.ENGINES.*enginename*.*projectname*.*queryname*.*windowname*.IN" and injects them into source windows.

As a result of the bus connectivity provided by the Tervela Data Fabric connector, the SAS Event Stream Processing engine does not need to manage individual publish/subscribe connections. A high capacity of concurrent publish/subscribe connections to a single ESP engine is achieved.

Note: Tervela functionality is not available on HP Itanium, AIX, SPARC, or 32–bit Microsoft Windows platforms.

You must install the Tervela run-time libraries on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (specify **`LD_LIBRARY_PATH`** on Linux platforms, for example).

The Tervela Data Fabric Connector has the following characteristics:

- It works with binary event blocks. No other event block formats are supported.
- It operates as a Tervela client. All Tervela Data Fabric connectivity parameters are required as connector configuration parameters.

Before using Tervela Data Fabric connectors, you must configure the following items on the Tervela TPM Provisioning and Management System:

- a client user name and password to match the connector's **`tvauserid`** and **`tvapassword`** configuration parameters
- the inbound and outbound topic strings and associated schema
- publish or subscribe entitlement rights associated with a client user name

When the connector starts, it publishes a message to topic SAS.META.*tvaclientname* (where *tvaclientname* is a connector configuration parameter). This message contains the following information:

- The mapping of the ESP engine name to a *host*:*port* field potentially used by an ESP publish/subscribe client. The *host*:*port* string is the required **`urlhostport`** connector configuration parameter, and is substituted by the engine name in topic strings used on the fabric.
- The project, query, and window names of the window associated with the connector, as well as the serialized schema of the window.

All messaging performed by the Tervela connector uses the Tervela Guaranteed Delivery mode. Messages are persisted to a Tervela TPE appliance. When a publisher connector connects to the fabric, it receives messages already published to the subscribed topic over a recent time period. By default, the publisher connector sets this time period to eight hours. This enables a publisher to catch up with a day's worth of

messages. Using this mode requires regular purging of persisted data by an administrator when there are no other automated mechanism to age out persisted messages.

Tervela subscriber connectors support a hot failover mode. The active/standby status of the connector is coordinated with the fabric so that a standby connector becomes active when the active connector fails. Several conditions must be met to guarantee successful switchovers:

- The engine names of the ESP engines running the involved connectors must all be identical. This set of ESP engines is called the failover group.
- All involved connectors must be active on the same set of topics.
- All involved subscriber connectors must be configured with the same **`tvaclientname`**.
- All involved connectors must initiate message flow at the same time, and with the TPE purged of all messages on related topics. This is required because message IDs must be synchronized across all connectors.
- Message IDs that are set by the injector of event blocks into the model must be sequential and synchronized with IDs used by other standby connectors. When the injector is a Tervela publisher connector, that connector sets the message ID on all injected event blocks, beginning with ID = 1.

When a new subscriber connector becomes active, outbound message flow remains synchronized due to buffering of messages by standby connectors and coordination of the resumed flow with the fabric. The size of this message buffer is a required parameter for subscriber connectors.

Tervela connector configuration parameters named “tva…” are passed unmodified to the Tervela API by the connector. See your Tervela documentation for more information about these parameters.

Use the following parameters with Tervela connectors:

Table 11.14 *Required Parameters for Subscriber Tervela Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`tvauserid`	Specifies a user name defined in the Tervela TPM. Publish-topic entitlement rights must be associated with this user name.
`tvapassword`	Specifies the password associated with **`tvauserid`**.
`tvaprimarytmx`	Specifies the host name or IP address of the primary TMX.
`tvatopic`	Specifies the topic name for the topic to which to subscribed. This topic must be configured on the TPM for the GD service and **`tvauserid`** must be assigned the Guaranteed Delivery subscribe rights for this Topic in the TPM.
`tvaclientname`	Specifies the client name associated with the Tervela Guaranteed Delivery context. If hot failover is enabled, this name must match the **`tvaclientname`** of other subscriber connectors in the failover group. Otherwise, the name must be unique among all instances of Tervela connectors.

Parameter	Description
tvamaxoutstand	Specifies the maximum number of unacknowledged messages that can be published to the Tervela fabric (effectively the size of the publication cache). Should be twice the expected transmit rate.
numbufferedmsgs	Specifies the maximum number of messages buffered by a standby subscriber connector. When exceeded, the oldest message is discarded. If the connector goes active the buffer is flushed, and buffered messages are sent to the fabric as required to maintain message ID sequence.
hotfailover	Enables hot failover mode
urlhostport	Specifies the "*host*/*port*" string sent in the metadata message published by the connector on topic SAS.META.*tvaclientname* when it starts.

Table 11.15 *Required Parameters for Publisher Tervela Connectors*

Parameter	Description
type	Specifies to publish.
tvauserid	Specifies a user name defined in the Tervela TPM. Subscribe-topic entitlement rights must be associated with this user name.
tvapassword	Specifies the password associated with **tvauserid**.
tvaprimarytmx	Specifies the host name or IP address of the primary TMX.
tvatopic	Specifies the topic name for the topic to which to publish. This topic must be configured on the TPM for the GD service.
tvaclientname	Specifies the client name associated with the Tervela Guaranteed Delivery context. Must be unique among all instances of Tervela connectors.
tvasubname	Specifies the name assigned to the Guaranteed Delivery subscription being created. The combination of this name and **tvaclientname** are used by the fabric to replay the last subscription state. If a subscription state is found, it is used to resume the subscription from its previous state. If not, the subscription is started new, starting with a replay of messages received in the past eight hours.
urlhostport	Specifies the "*host*:*port*" string sent in the metadata message published by the connector on topic SAS.META.*tvaclientname* when it starts.

Table 11.16 *Optional Parameters for Subscriber Tervela Connectors*

Parameter	Description
snapshot	Disables the sending of snapshot data. The default value is enabled.
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
tvasecondarytmx	Specifies the host name or IP address of the secondary TMX. Required if logging in to a fault-tolerant pair.
tvalogfile	Causes the connector to log to the specified file instead of to syslog (on Linux or Solaris) or Tervela.log (on Windows)
tvapubbwlimit	Specifies the maximum bandwidth, in Mbps, of data published to the fabric. The default value is 100 Mbps.
tvapubrate	Specifies the rate at which data messages are published to the fabric, in Kbps. The default value is 30,000 messages per second.
tvapubmsgexp	Specifies the maximum amount of time, in seconds, that published messages are kept in the cache in the Tervela API. This cache is used as part of the channel egress reliability window (if retransmission is required). The default value is 1 second.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.

Table 11.17 *Optional Parameters for Publisher Tervela Connectors*

Parameter	Description
tvasecondarytmx	Specifies the host name or IP address of the secondary TMX. Required when logging in to a fault-tolerant pair.
tvalogfile	Causes the connector to log to the specified file instead of to syslog (on Linux or Solaris) or Tervela.log (on Windows)

Using the Solace Systems Connector

The Solace Systems connector communicates with a hardware-based Solace fabric for publish and subscribe operations.

A Solace subscriber connector receives event blocks and publishes them to this Solace topic:

"*host*:*port*/*projectname*/*queryname*/*windowname*/O

A Solace publisher connector reads event blocks from the following Solace topic

"*host*:*port*/*projectname*/*queryname*/*windowname*/I

and injects them into the corresponding source window.

As a result of the bus connectivity provided by the connector, the SAS Event Stream Processing engine does not need to manage individual publish/subscribe connections. A high capacity of concurrent publish/subscribe connections to a single ESP engine is achieved.

Note: Solace functionality is not available on HP Itanium, AIX, or 32–bit Microsoft Windows platforms.

The Solace run-time libraries must be installed on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (for example, specifying **`LD_LIBRARY_PATH`** on Linux platforms).

The Solace Systems connector has the following characteristics:

- By default, it works with binary event blocks. Alternatively, you can specify the **`protofile`** and **`protomsg`** configuration parameters to enable transport of event blocks that are encoded as Google protocol buffer messages.
- It operates as a Solace client. All Solace connectivity parameters are required as connector configuration parameters.

You must configure the following items on the Solace appliance to which the connector connects:

- a client user name and password to match the connector's **`soluserid`** and **`solpassword`** configuration parameters
- a message VPN to match the connector's **`solvpn`** configuration parameter
- On the message VPN, you must enable "Publish Subscription Event Messages".
- On the message VPN, you must enable "Client Commands" under "SEMP over Message Bus".
- On the message VPN, you must configure a nonzero "Maximum Spool Usage".
- When hot failover is enabled on subscriber connectors, you must create a single exclusive queue named "active_esp" in the message VPN. The subscriber connector that successfully binds to this queue becomes the active connector.
- When *buspersistence* is enabled, you must enable "Publish Client Event Messages" on the message VPN.
- When *buspersistence* is enabled, you must create exclusive queues for all subscribing clients. The exclusive queue name must be the same as that specified for the *buspersistencequeue* parameter.

When the connector starts, it subscribes to topic "*urlhostport*/M" (where *urlhostport* is a connector configuration parameter). This enables the connector to receive metadata requests from clients that publish or subscribe to a window in an ESP engine associated with that *host*:*port* combination. Metadata responses consist of some combination of the project, query, and window names of the window associated with the connector, as well as the serialized schema of the window.

Solace subscriber connectors support a hot failover mode. The active/standby status of the connector is coordinated with the fabric so that a standby connector becomes active when the active connector fails. Several conditions must be met to guarantee successful switchovers:

- All involved connectors must be active on the same set of topics.
- All involved connectors must initiate message flow at the same time. This is required because message IDs must be synchronized across all connectors.

- Google protocol buffer support must not be enabled, because these binary messages do not contain a usable message ID.

When a new subscriber connector becomes active, outbound message flow remains synchronized due to buffering of messages by standby connectors and coordination of the resumed flow with the fabric. The size of this message buffer is a required parameter for subscriber connectors.

Solace connectors can be configured to use a persistent mode of messaging instead of the default direct messaging mode. (See the description of the **buspersistence** configuration parameter.) This mode might require regular purging of persisted data by an administrator, if there are no other automated mechanism to age out persisted messages. The persistent mode reduces the maximum throughput of the fabric, but it enables a publisher connector to connect to the fabric after other connectors have already processed data. The fabric updates the connector with persisted messages and synchronizes window states with the other engines in a hot failover group.

Solace subscriber connectors subscribe to a special topic that enables them to be notified when a Solace client subscribes to the connector's topic. When the connector is configured with snapshot enabled, it sends a custom snapshot of the window contents to that client. This enables late subscribers to catch up upon connecting.

Solace connector configuration parameters named "sol…" are passed unmodified to the Solace API by the connector. See your Solace documentation for more information about these parameters.

Use the following parameters with Solace Systems connectors

Table 11.18 *Required Parameters for Subscriber Solace Connectors*

Parameter	Description
type	Specifies to subscribe.
soluserid	Specifies the user name required to authenticate the connector's session with the appliance.
solpassword	Specifies the password associated with **soluserid**.
solhostport	Specifies the appliance to connect to, in the form "*host*:*port*".
solvpn	Specifies the appliance message VPN to assign the client to which the session connects.
soltopic	Specifies the Solace destination topic to which to publish.
urlhostport	Specifies the *host*:*port* field in the metadata topic subscribed to on start-up to field metadata requests.
numbufferedmsgs	Specifies the maximum number of messages buffered by a standby subscriber connector. If exceeded, the oldest message is discarded. If the connector goes active the buffer is flushed, and buffered messages are sent to the fabric as required to maintain message ID sequence.
hotfailover	Enables hot failover mode.

Table 11.19 *Required Parameters for Publisher Solace Connectors*

Parameter	Description
`type`	Specifies to publish.
`soluserid`	Specifies the user name required to authenticate the connector's session with the appliance.
`solpassword`	Specifies the password associated with **`soluserid`**.
`solhostport`	Specifies the appliance to connect to, in the form "*host*:*port*".
`solvpn`	Specifies the appliance message VPN to assign the client to which the session connects.
`soltopic`	Specifies the Solace topic to which to subscribe.
`urlhostport`	Specifies the *host*:*port* field in the metadata topic subscribed to on start-up to field metadata requests.

Table 11.20 *Optional Parameters for Subscriber Solace Connectors*

Parameter	Description
`snapshot`	Disables the sending of snapshot data. The default is enabled.
`collapse`	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
`buspersistence`	Sets the Solace message delivery mode to Guaranteed Messaging. The default value is Direct Messaging.
`rmretdel`	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
`protofile`	Specifies the **`.proto`** file that contains the Google Protocol Buffers message definition used to convert event blocks to **`protobuf`** messages. When you specify this parameter, you must also specify the **`protomsg`** parameter.
`protomsg`	Specifies the name of a Google Protocol Buffers message in the **`.proto`** file that you specified with the **`protofile`** parameter. Event blocks are converted into this message.

Table 11.21 *Optional Parameters for Publisher Solace Connectors*

Parameter	Description
buspersistence	Creates the Guaranteed message flow to bind to the topic endpoint provisioned on the appliance that the published Guaranteed messages are delivered and spooled to. By default this flow is disabled, because it is not required to receive messages published using Direct Messaging.
buspersistencequeue	Specifies the name of the exclusive queue to which the Guaranteed message flow binds.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition used to convert event blocks to **protobuf** messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.

Using the Tibco Rendezvous (RV) Connector

The Tibco Rendezvous (RV) connector supports the Tibco RV API for publish and subscribe operations through a Tibco RV daemon. The subscriber receives event blocks and publishes them to a Tibco RV subject. The publisher is a Tibco RV subscriber, which injects received event blocks into source windows.

The Tibco RV run-time libraries must be installed on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (for example, specifying **LD_LIBRARY_PATH** on Linux platforms).

The system path must point to the **Tibco/RV/bin** directory so that the connector can run the RVD daemon.

The subject name used by a Tibco RV connector is a required connector parameter. A Tibco RV subscriber also requires a parameter that defines the message format used to publish events to Tibco RV. The format options are CSV or binary. A Tibco RV publisher can consume any message type produced by a Tibco RV subscriber.

By default, the Tibco RV connector assumes that a Tibco RV daemon is running on the same platform as the connector. Alternatively, you can specify the connector **tibrvdaemon** configuration parameter to use a remote daemon.

Similarly, you can specify the optional **tibrvservice** and **tibrvnetwork** parameters to control the Rendezvous service and network interface used by the connector. For more information, see your Tibco RV documentation.

The Tibco RV connector relies on the default multicast protocols for message delivery. The reliability interval for messages sent to and from the Tibco RV daemon is inherited from the value in use by the daemon.

Use the following parameters with Tibco RV connectors:

Table 11.22 *Required Parameters for Subscriber Tibco RV Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`tibrvsubject`	Specifies the Tibco RV subject name.
`tibrvtype`	Specifies binary or CSV.

Table 11.23 *Required Parameters for Publisher Tibco RV Connectors*

Parameter	Description
`type`	Specifies to publish.
`tibrvsubject`	Specifies the Tibco RV subject name.

Table 11.24 *Optional Parameters for Subscriber Tibco RV Connectors*

Parameter	Description
`snapshot`	Disables the sending of snapshot data. The default value is enabled.
`collapse`	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
`tibrvservice`	Specifies the Rendezvous service used by the Tibco RV transport created by the connector. The default service name is “rendezvous”.
`tibrvnetwork`	Specifies the network interface used by the Tibco RV transport created by the connector. The default network depends on the type of daemon used by the connector.
`tibrvdaemon`	Specifies the Rendezvous daemon used by the connector. The default is the default socket created by the local daemon.
`rmretdel`	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.

Table 11.25 *Optional Parameters for Publisher Tibco RV Connectors*

Parameter	Description
`blocksize`	Specifies the number of events to include in a published event block. The default value is 1.

Parameter	Description
transactional	Sets the event block type to transactional. The default value is normal.
tibrvservice	Specifies the Rendezvous service used by the Tibco RV transport created by the connector. The default service name is “rendezvous”.
tibrvnetwork	Specifies the network interface used by the Tibco RV transport created by the connector. The default network depends on the type of daemon used by the connector.
tibrvdaemon	Specifies the Rendezvous daemon used by the connector. The default is the default socket created by the local daemon.

Using the PI Connector

The PI Connector supports publish and subscribe operations for a PI Asset Framework (AF) server. The SAS Event Stream Processing Engine model must implement a window of a fixed schema to carry values that are associated with AF attributes owned by AF elements in the AF hierarchy. The AF data reference can be defined as a PI Point for these elements.

Note: Support for the PI Connector is available only on 64–bit Microsoft Windows platforms.

The PI Asset Framework (PI AF) Client from OSIsoft must be installed on the Microsoft Windows platform that hosts the running instance of the connector. The connector loads the OSIsoft.AFSDK.DLL public assembly, which requires .NET 4.0 installed on the target platform. The run-time environment must define the path to OSIsoft.AFSDK.dll.

The Microsoft Visual C++ Redistributable for Visual Studio 2012 package must be installed on the Microsoft Windows platform. The library for this connector is built with a different version of tools than those used by SAS Event Stream Processing Engine libraries in order to achieve .NET 4.0 compatibility.

The SAS Event Stream Processing Engine window that is associated with the connector must use the following schema:

```
ID*:int32,elementindex:string,element:string,attribute:string,value:variable:,
timestamp:stamp,status:string
```

Use the following schema values.

Schema Value	Description
elementindex	Value of the connector *afelement* configuration parameter. Specify either an AF element name or an AF element template name.
element	Name of the AF element associated with the *value*.
attribute	Name of the AF attribute owned by the AF *element*.

Schema Value	Description
value	Value of the AF *attribute*.
timestamp	Timestamp associated with the *value*.
status	Status associated with the *value*.

Note: The type of the *value* field is determined by the type of the AF attribute as defined in the AF hierarchy.

The following mapping of event stream processor data type to AF attributes is supported:

Event Stream Processor Data Type	AF Attribute
ESP_DOUBLE	TypeCode::Single TypeCode::Double
ESP_INT32	TypeCode::Byte TypeCode::SByte TypeCode::Char TypeCode::Int16 TypeCode::UInt16 TypeCode::Int32 TypeCode::UInt32
ESP_INT64	TypeCode::Int64 TypeCode::UInt64
ESP_UTF8STR	TypeCode::String
ESP_TIMESTAMP	TypeCode::DateTime

If an attribute has **`TypeCode::Object`**, the connected uses the type of the underlying PI point. Valid event stream processing data types to PI point mappings are as follows:

Event Stream Processor Data Type	PI Point
ESP_INT32	PIPointType::Int16 PIPointType::Int32
ESP_DOUBLE	PIPointType::Float16 PIPointType::Float32
ESP_TIMESTAMP	PIPointType::Timestamp

Event Stream Processor Data Type	PI Point
ESP_UTF8STR	PIPointType::String

Use the following parameters with PI connectors:

Table 11.26 *Required Parameters for Subscriber PI Connectors*

Parameter	Description
type	Specifies to subscribe.
afelement	Specifies the AF element or element template name. Wildcards are supported.
iselementtemplate	Specifies whether the **afelement** parameter is an element template name. By default, the **afelement** parameter specifies an element name. Valid values are **TRUE** or **FALSE**.

Table 11.27 *Required Parameters for Publisher PI Connectors*

Parameter	Description
type	Specifies to publish.
afelement	Specifies the AF element or element template name. Wildcards are supported.
iselementtemplate	Specifies that the **afelement** parameter is an element template name. By default, the **afelement** parameter specifies an element name.

Table 11.28 *Optional Parameters for Subscriber PI Connectors*

Parameter	Description
snapshot	Disables sending of snapshot data. The default is enabled.
rmretdel	Remove all delete events from event blocks received by the subscriber that were introduced by a window retention policy.
pisystem	Specifies the PI system. The default is the PI system that is configured in the PI AF client.
afdatabase	Specifies the AF database. The default is the AF database that is configured in the PI AF client.
afrootelement	Specifies the root element in the AF hierarchy from which to search for parameter **afelement**. The default is the top-level element in the AF database.

Parameter	Description
afattribute	Specifies a specific attribute in the element. The default is all attributes in the element.

Table 11.29 *Optional Parameters for Publisher PI Connectors*

Parameter	Description
blocksize	Specifies the number of events to include in a published event block. The default value is 1.
transactional	Sets the event block type to transactional. The default value is normal.
pisystem	Specifies the PI system. The default is the PI system that is configured in the PI AF client.
afdatabase	Specifies the AF database. The default is the AF database that is configured in the PI AF client.
afrootelement	Specifies the root element in the AF hierarchy from which to search for the parameter **afelement**. The default is the top-level element in the AF database.
afattribute	Specifies a specific attribute in the element. The default is all attributes in the element.
archivetimestamp	Specifies that all archived values from the specified timestamp onwards are to be published when connecting to the PI system. The default is to publish only new values.

Using the Teradata Connector

The Teradata connector uses the Teradata Parallel Transporter (TPT) API to support subscribe operations against a Teradata server.

The Teradata Tools and Utilities (TTU) package must be installed on the platform running the connector. The run-time environment must define the path to the Teradata client libraries in the TTU. For Teradata ODBC support while using the load operator, you must also define the path to the Teradata TeraGSS libraries in the TTU. To support logging of Teradata defined messages, you must define the **NLSPATH** environment variable appropriately. For example, with a TTU installation in **/opt/teradata**, define **NLSPATH** as "**/opt/teradata/client/*version*/tbuild/msg64/%N**". The connector has been certified using TTU version 14.10.

For debugging, you can install optional PC-based Teradata client tools to access the target database table on the Teradata server. These tools include the GUI SQL workbench (Teradata SQL Assistant) and the DBA/Admin tool (Teradata Administrator), which both use ODBC.

You must use one of three TPT operators to write data to a table on the server:**stream**, **update**, or **load**.

Operator	Description
stream	Works like a standard ESP database subscriber connector, but with improved throughput gained through using the TPT. Supports insert, update, and delete events. As it receives events from the subscribed window, it writes them to the target table. If you set the required **tdatainsertonly** configuration parameter to **false**, serialization is automatically enabled in the TPT to maintain correct ordering of row data over multiple sessions.
update	Supports insert/update/delete events but writes them to the target table in batch mode. The batch period is a required connector configuration parameter. At the cost of higher latency, this operator provides better throughput with longer batch periods (for example minutes instead of seconds).
load	Supports insert events. Requires an empty target table. Provides the most optimized throughput. Staggers data through a pair of intermediate staging tables. These table names and connectivity parameters are additional required connector configuration parameters. In addition, the write from a staging table to the ultimate target table uses the generic ODBC driver used by the database connector. Thus, the associated connect string configuration and odbc.ini file specification is required. The staging tables are automatically created by the connector. If the staging tables and related error and log tables already exist when the connector starts, it automatically drops them at start-up.

Table 11.30 *Required Parameters for Teradata Connectors*

Parameter	Description
type	Specifies to subscribe
desttablename	Target table name.
tdatausername	User name for the user account on the target Teradata server.
tdatauserpwd	User password for the user account on the target Teradata server.
tdatatdpid	Target Teradata server name
tdatamaxsessions	Maximum number of sessions created by the TPT to the Teradata server.
tdataminsessions	Minimum number of sessions created by the TPT to the Teradata server.
tdatadriver	The operator: **stream**, **update**, or **load**.
tdatainsertonly	Specifies whether events in the subscriber event stream processing window are insert only. Must be **true** when using the load operator.

Table 11.31 *Optional Parameters for Teradata Connectors*

Parameter	Description
snapshot	Disables the sending of snapshot data. By default, sending snapshot data is enabled.
rmretdel	Removes all delete events from event blocks received by the subscriber that were introduced by a window retention policy.
tdatabatchperiod	Specifies the batch period in seconds. Required when using the **update** operator, otherwise ignored.
stage1tablename	Specifies the first staging table. Required when using the **load** operator, otherwise ignored.
stage2tablename	Specifies the second staging table. Required when using the **load** operator, otherwise ignored.
connectstring	Specifies the connect string used to access the target and staging tables. Use the form **"DSN=*dsnname*;UID=*userid*;pwd=*password*"**. Required when using the **load** operator, otherwise ignored.
tdatatracelevel	Specifies the trace level for Teradata messages written to the trace file in the current working directory. The trace file is named *operator*1.txt. Default is 1 (TD_OFF). Other valid values are: 2 (TD_OPER), 3 (TD_OPER_CLI), 4 (TD_OPER_OPCOMMON), 5 (TD_OPER_SPECIAL), 6 (TD_OPER_ALL), 7 (TD_GENERAL), and 8 (TD_ROW).

Writing a Custom Connector

When you write your own connector, the connector class must inherit from base class **dfESPconnector**.

Connector configuration is maintained in a set of key or value pairs where all keys and values are text strings. A connector can obtain the value of a configuration item at any time by calling **getParameter()** and passing the key string. An invalid request returns an empty string.

A connector can implement a subscriber that receives events generated by a window, or a publisher that injects events into a window. However, a single instance of a connector cannot publish and subscribe simultaneously.

A subscriber connector receives events by using a callback method defined in the connector class that is invoked in a thread owned by the engine. A publisher connector typically creates a dedicated thread to read events from the source. It then injects those events into a source window, leaving the main connector thread for subsequent calls made into the connector.

A connector must define these static data structures:

Static Data Structure	Description
dfESPconnectorInfo	Specifies the connector name, publish/subscribe type, initialization function pointer, and configuration data pointers.
subRequiredConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing required configuration parameters for a subscriber.
sizeofSubRequiredConfig	Specifies the number of entries in **subRequiredConfig**.
pubRequiredConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing required configuration parameters for a publisher.
sizeofPubRequiredConfig	Specifies the number of entries in **pubRequiredConfig**.
subOptionalConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing optional configuration parameters for a subscriber.
sizeofSubOptionalConfig	Specifies the number of entries in **subOptionalConfig**.
pubOptionalConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing optional configuration parameters for a publisher.
sizeofPubOptionalConfig	Specifies the number of entries in **pubOptionalConfig**.

A connector must define these static methods:

Static Method	Description
dfESPconnector *initialize(dfESPengine *_engine_, dfESPpsLib_t _psLib_)	Returns an instance of the connector.
dfESPconnectorInfo *getConnectorInfo()	Returns the **dfESPconnectorInfo** structure.

You can invoke these static methods before you create an instance of the connector.

A connector must also define these virtual methods:

Virtual Method	Description
start()	Starts the connector. Must call base class method **checkConfig()** to validate connector configuration before starting. Must also call base class method **start()**. Must set **variable _started = true** upon success.
stop()	Stops the connector. Must call base class method **stop()**. Must leave the connector in a state whereby **start()** can be subsequently called to restart the connector.
getState()	Returns the current state of the connector.
callbackFunction()	Specifies the method invoked by the engine to pass event blocks generated by the window to which it is connected.
errorCallbackFunction()	Specifies the method invoked by the engine to report errors detected by the engine. Must call user callback function **errorCallback**, if nonzero.

Finally, a derived connector can implement up to ten user-defined methods that can be called from an application. Because connectors are plug-ins loaded at run time, a user application cannot directly invoke class methods. It is not linked against the connector.

The base connector class defines virtual methods **userFunction_01** through **userFunction_10**, and a derived connector then implements those methods as needed. For example:

```
void * myConnector::userFunction_01(void *myData) {
```

An application would invoke the method as follows:

```
myRC = myConnector->userFunction_01((void *)myData);
```

Integrating a Custom Connector

All connectors are managed by a global connector manager. The default connectors shipped with SAS Event Stream Processing Engine are automatically loaded by the connector manager during product initialization. Custom connectors built as libraries and placed in **$DFESP_HOME/lib/plugins** are also loaded during initialization, with the exception of those listed in **$DFESP_HOME/etc/connectors.excluded**.

After initialization, the connector is available for use by any event stream processor window defined in an application. As with any connector, an instance of it can be obtained by calling the window **getConnector()** method and passing its user-defined method. You can configure the connector using **setParameter()** before starting the project.

Chapter 12

Using Adapters

Overview to Adapters

Adapters are stand-alone executable files that use the publish/subscribe API to do the following:

- publish event streams into an engine
- subscribe to event streams from engine windows

Unlike connectors, which are built into the SAS Event Stream Processing Engine, adapters can be networked. Many adapters are executable versions of connectors. Thus, the required and optional parameters of most adapters directly map to the parameters of the corresponding connector.

You can find adapters in the following directory:

```
$DFESP_HOME/bin
```

Using the SAS LASR Analytic Server Adapter

The SAS LASR Analytic Server adapter supports publishing insert-only data to the SAS LASR Analytic Server. To run this adapter in a UNIX requirement requires the installation of an SSH client. Do not run this adapter in a Microsoft Windows environment.

Use the following form for both publish and subscribe use:

dfesp_lasr_adpater-h *url*-d *LASRhost:LASRport* -t *LASRtablename* <-a *blockstocommit*> <-b *buffersize*> <-s *fields*>

Parameter	Description
-h *url*	Specify the publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window*** Append the following for subscribers: **?snapshot=true | false**.
-d *LASRhost:LASRport*	Specify the SAS LASR Analytic Server *hostname* and *port*.
-t *LASRtablename*	Specify the name of the LASR table.
-a *blockstocommit*	Specify the number of blocks to commit. The default value is 8. This parameter trades throughput for latency. The larger the value, the more event blocks are appended to the LASR table before being committed.
-b *buffersize*	Specify the buffer size. The default value is 512. This buffering parameter specifies the number of rows that the adapter buffers before sending them to the SAS LASR Analytic Server.
-s *fields*	Specify the fields to send to the SAS LASR Analytic Server. Use the form *f1_name,...fn_name*

Note:

- A table in the SAS LASR Analytic Server must exist, and its columns should match a subset of the columns in the event stream processor.
- When the column names in the LASR table match all the fields names in the event stream processing window, you do not need to use the **-s** parameter. The event stream processing schema corresponds one–to–one with the LASR schema. All LASR columns are populated from all event stream processing window columns.
- When the LASR table contains fewer columns than the event stream processing window, use **-s *field_1, ... ,field_n*** to choose the subset of event stream processing fields to send to the LASR table.

Using the File and Socket Adapter

The File and Socket adapter supports publish and subscribe operations on files or socket connections that stream the following data types:

- dfESP binary
- CSV
- XML
- JSON
- syslog
- hdat

Subscriber use:

dfesp_fs_adapter -k sub - h *url* -f *fsname*-t binary | csv | xml | json | hdat
<-c *period*> <-s *maxfilesize*> <-d *dateformat*>
<-r *rate*> <-a *aggrsize*> <-n>
<-g *gdconfig*> <-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off> <-y *log_configfile*> <-o *hdat_filename*>
<-q *hdat_max_data_nodes*> <-u *hdfs_blocksize*> <-v *hdfs_numreplicas*>
<-w *hdat_numthreads*> <-z *hdat_max_stringlength*>

Publisher use:

dfesp_fs_adapter-k pub -h *url* -f *fsname*
-t binary | csv | xml | json | syslog <-b *blocksize*> <-d *dateformat*>
<-r *rate*> <-m *maxevents*> <-e> <-i> <-p> <-n> <-g *gdconfig*>
<-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal | off>
<-y *log_configfile*> <-x *header*> <-Q>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
–h *url*	Specify the dfESP publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window*** Append the following for subscribers: **?snapshot=true** | **false**. Append the following for subscribers if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
–f *fsname*	Specify the subscriber output file, publisher input file, or socket “***host*:*port***". Leave *host* blank to implement a server.
-t binary | **csv** | **xml** | **json** | **syslog** | **hdat**	Specify the file system type. The **syslog** value is valid only for publishers. The **hdat** value is valid only for subscribers.

Parameter	Definition
–c ***period***	Specify output file time (in seconds). The active file is closed and a new active file opened with the timestamp appended to the filename
-s ***maxfilesize***	Specify the output file volume (in bytes). The active file is closed and a new active file opened with the timestamp appended to the filename.
–d ***dateformat***	Specify the date format. The default value = %Y-%m-%d %H:%M:%S
-r ***rate***	Specify the requested transmit rate in events per second.
–a ***aggrsize***	Specify, in latency mode, statistics for aggregation block size.
–n	Specify latency mode.
-g ***gdconfig***	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
-b ***blocksize***	Set the block size. The default value = 1.
–m ***maxevents***	Specify the maximum number of events to publish.
-e	Specify that events are transactional.
-i	Read from growing file.
-p	Buffer all event blocks before publishing them.
–y ***logconfigfile***	Specify the log configuration file.
-x ***header***	Specify the number of header lines to ignore.
-o ***hdat_filename***	Specify the filename to write to Hadoop Distributed File System (HDFS).
-q ***hdat_max_datanodes***	Specify the maximum number of data nodes. The default value is the number of live data nodes.
-u ***hdfs_blocksize***	Specify the Hadoop Distributed File System (HDFS) block size in MB.

Parameter	Definition
-v ***hdfs_numreplicas***	Specify the Hadoop Distributed File System (HDFS) number of replicas. The default value is 1.
-w ***hdat_numthreads***	Specify the adapter thread pool size. A value <= 1 means that no data nodes are used.
-z ***hdat_max_stringlength***	Specify the fixed size to use for string fields in HDAT records. The value must be a multiple of 8.
-Q	For the publisher, quiesce the project after all events are injected into the source window.

If you specify Solace or Tervela transports instead of the default native transport, refer to the description of the C++ **`C_dfESPpubsubSetPubsubLib()`** API call for more information about the required client configuration files.

Using the Database Adapter

The Database Adapter supports publish and subscribe operations on databases using DataDirect drivers. The adapter is certified for the following platforms

- Oracle
- MySQL
- IBM DB2
- Greenplum
- PostgreSQL
- SAP Sybase ASE
- Teradata
- Microsoft SQL Server
- IBM Informix
- Sybase IQ

However, drivers exist for many other databases and appliances. This adapter requires that database connectivity be available by using a Data Source Name (DSN) configured in the odbc.ini file. The file is pointed to by the ODBCINI environment variable.

Note: The database adapter uses generic DataDirect ODBC drivers for the Teradata platform. A separately packaged adapter uses the Teradata Parallel Transporter for improved performance.

Subscriber usage:

dfesp_db_adapter -k sub -h *url* -d *DSNname* -u *userid*
-x *pwd* -t *tablename* <-g *gdconfig*> <-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*>

Publisher usage:

dfesp_db_adapter -k pub —h *host* -d *DSNname*
-u *userid* -x *pwd* -s *selectstatement*
<-b *blocksize*> <-e> <-g *gdconfig*> <-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
—h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***. Append **?snapshot=true** | **false** if needed. Append **?rmretdel=true** | **false** for subscribers if needed.
—d *DSNname*	Specify the data source name from the odbc.ini file.
—u *userid*	Specify the database user ID.
—x *pwd*	Specify the database user password.
-t *tablename*	Specify the subscriber target database table.
—s *selectstatement*	Specify the publisher source database SQL statement.
-b *blocksize*	Specify the block size. The default value = 1.
—l native | solace | tervela	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
—j trace | debug | info | warn | error | fatal | off	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
-e	Specify that events are transactional.
—g *gdconfig*	Specify the guaranteed delivery configuration file.
—y *logconfigfile*	Specify the log configuration file.

When you specify Solace or Tervela transports instead of the default native transport, client configuration files are required. Refer to the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call for more information these files.

Using the SMTP Subscriber Adapter

The SMTP Subscriber Adapter subscribes to only SAS Event Stream Processing Engine windows. Publishing to a source window is not supported. This adapter forwards subscribed event blocks or single events as e-mail messages to a configured SMTP server, with the event data in the message body encoded in CSV format.

Subscriber use:

dfesp_smtp_adapter-h *url*-m *smtpserver* -u *sourceaddress*
-d *destaddress* <-p> <-g *gdconfig*> <-l native | solace | tervela >
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*>

Parameter	Definition
-h ***url***	Specify the dfESP subscribe standard URL in the form "**dfESP://** ***host:port/project/continuousquery/window*?** **snapshot=true** |**false**. You can append the following if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
-p	Specify that each e-mail contains a single event instead of a complete event block containing one or more events.
-g	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
-y ***logconfigfile***	Specify the log configuration file.

Using the Event Stream Processor to Event Stream Processing Engine Adapter

The event stream processor to SAS Event Stream Processing Engine adapter enables you to subscribe to a window and publish what it passes into another source window. This operation can occur within a single event stream processor. More likely it occurs across two event stream processors that are run on different machines on the network.

No corresponding event stream processor to SAS Event Stream Processing Engine connector is provided.

Usage:

dfesp_esp_adapter -s *url* –p *url*

Parameter	Definition
-s *url*	Specify the subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*contquery*/*window*?snapshot=true |false>?collapse=true | false**"
-p *url*	Specify the publish standard URL in the form **dfESP://*host*:*port*/*project*/*contquery*/*window***

The **eventblock** size and transactional nature is inherited from the subscribed window.

Using the SAS Data Set Subscriber Adapter

The SAS Data Set Subscriber Adapter supports subscribe, but it does not support publish. It captures subscribed event blocks and writes them to an output SAS data set file on the file system where the adapter is running.

This adapter has no corresponding connector.

The adapter consists of two entities:

- **dfesp_dataset.sas**: This SAS script takes event data in CSV format and writes it to the SAS data set. It receives CSV data by executing **subscribe_client** and intercepting console output.
- **subscribe_client**: This subscriber client handles the connection to the event stream processor. It outputs event data to the console in a format and sequence required by **dfesp_dataset.sas**.

The **dfesp_dataset.sas** script requires an installed SAS system and a SAS spawner running on the host-port endpoint specified in **dfesp_dataset.sas**. To determine whether the SAS spawner is running, execute the following command:

```
netstat -tulpn | fgrep <port>
```

To start the SAS spawner, add **unxspawn** on ***port*/tcp** to your **/etc/services** file. Then execute the following command from your **/SASFoundation/x.y/utilities/bin** directory:

```
./sastcpd -service unxspawn -sascmd your_SAS_user_directory/signon &
```

To set the spawner host-port used by **dfesp_dataset.sas**, find the **let tsk** statement and modify it as follows:

```
%let tsk&c=host port;
```

To set the user credentials, find the **let tcpsec** statement and modify it as follows:

```
%let tcpsec=userid.password;
```

You need to configure SAS environment variables that are required to run the subscriber client. Edit the .cfg file in your **/SASFoundation** directory and add these two statements:

```
-SET DFESP_HOME the home directory of your Event Stream Processing installation
-SET LD_LIBRARY_PATH the paths to your Event Stream Processing libraries
```

You can copy these settings directly from the environment variables already set for your installation.

Before running the **dfesp_dataset.sas** script, make sure your environment path includes the following:

- the path to your SAS installation (usually **/SASFoundation/x.y**)
- the path to the **subscribe_client** executable

To run **dfesp_dataset.sas**, execute the following command from the script's directory:

```
sas dfesp_dataset.sas -sysparm outputpath url"
```

where

- *outputpath* specifies where the SAS output data set and the SAS log file are written. The data set filename consists of the subscribed window name appended with a timestamp.
- *url* specifies where the engine is running and a window to subscribe to. Specify a standard event stream processing style URL in the following form:

  ```
  dfESP://host:port/project/contquery/window
  ```

You can provide multiple URLs. Each URL can specify a different subscribed window. In that case, a separate output data set is written for each specified URL.

Using the Java Message Service (JMS) Adapter

The Java Message Service (JMS) adapter resides in **dfx-esp-jms-adapter.jar**, which bundles the Java publisher and subscriber SAS Event Stream Processing Engine clients. Both are JMS clients. The subscriber client receives SAS Event Stream Processing Engine event blocks and is a JMS message producer. The publisher client is a JMS message consumer and injects event blocks into source windows of an SAS Event Stream Processing Engine.

The JMS topic string used by an event stream processor JMS client is passed as a required parameter to the adapter.

The subscriber client requires a command line parameter that defines the type of JMS message used to contain SAS Event Stream Processing Engine events. The publisher client consumes any message type produced by the subscriber client.

The client target platform must connect to a running **JMS_broker** (or JMS server) . The environment variable **DFESP_JMS_JARS** must specify the location of the JMS broker JAR files. The clients also require a **jndi.properties** file, which you must specify through the **DFESP_JMS_PROPERTIES** environment variable. This properties file specifies the connection factory that is needed to contact the broker and create JMS connections.

A sample jndi.properties file is included in the **etc** directory of the SAS Event Stream Processing Engine installation. Do not modify the **connectionFactoryNames** property, because the client classes look for that name.

Subscriber use on UNIX and Linux:

$DFESP_HOME/bin/dfesp_jms_subscriber -u *url* -j *jmstopic*
-m BytesMessage | TextMessage | MapMessage <-d *dateformat*>
<- g *gdconfigfile*><-l native | solace | tervela>
<-o severe | warning | info>

Subscriber use on Windows:

$DFESP_HOME\bin\dfesp_jms_subscriber *urljmstopic*
BytesMessage | TextMessage | MapMessage <*dateformat*> <*gdconfigfile* >
<native | solace | tervela>< severe | warning | info>

Parameter	Definition
– u ***url***	Specify the dfESP subscribe standard URL in the form **dfESP://** ***host:port/project/continuousquery/window***. Append the following for subscribers: **?snapshot=true** | **false**. Append the following for subscribers if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
–j ***jmstopic***	Specify the JMS topic name.
– d ***dateformat***	Specify the format of ESP_datetime_t and ESP_timestamp_t fields in events. The default is "yyyy-MM-dd HH:mm:ss.SSS".
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
- g ***gdconfigfile***	Specify the guaranteed delivery configuration file for the client.
-o severe | **warning** | **info**	Specify the application logging level.

Publisher use on UNIX and Linux:

$DFESP_HOME/bin/dfesp_jms_publisher -u *url* -j *jmstopic*
-b *blocksize* <-t> <-d *dateformat*>
<- g *gdconfigfile*> <-l native | solace | tervela> <-o severe | warning | info>

Publisher use on Windows:

$DFESP_HOME\bin\dfesp_jms_publisher *url jmstopic*
blocksize <transactional> <*dateformat*>
<*gdconfigfile*><native | solace | tervela >< severe | warning | info>

Parameter	Definition
–u ***url***	Specify the publish and subscribe standard URL in the form "**dfESP://***host***:***port***/***project***/***continuousquery***/***window***"
–j ***jmstopic***	Specify the JMS topic name.
–b ***blocksize***	Specify the number of events per event block.
-t	Specify that event blocks are transactional. The default is normal.
– d ***dateformat***	Specify the format of ESP_datetime_t and ESP_timestamp_t fields in events. The default is "yyyy-MM-dd HH:mm:ss.SSS".
- g ***gdconfigfile***	Specify the guaranteed delivery configuration file for the client.
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
-o severe | **warning** | **info**	Specify the application logging level.

Using the IBM WebSphere MQ Adapter

The MQ Adapter supports publish and subscribe operations on IBM WebSphere Message Queue systems. To use this adapter, you must install IBM WebSphere MQ Client run-time libraries and define the environment variable MQSERVER to specify the adapter's MQ connection parameters.

Subscriber usage:

dfesp_mq_adapter -k sub -h *url* —f *mqtopic*
<-q *mqqueuemanager*> <d *dateformat*> <-g *gdconfig*>
<-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*>

Publisher usage:

dfesp_mq_adapter -k pub -h *url* —f *mqtopic*
—n *mqsubname* —s *mqsubqueue* <-q *mqqueuemanager*>
<-b *blocksize*> <-d *dateformat*> <-e> <-g *gdconfig*>
<-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*>

Parameter	Definition
-k	Specify "sub" for subscriber use and "pub" for publisher use

Parameter	Definition
-h ***url***	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window*".**.** Append the following for subscribers: **?snapshot=true** | **false**. Append the following for subscribers if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
-f ***mqtopic***	Specify the MQ topic name.
-n ***mqsubname***	Specify the MQ subscription name.
-s ***mqsubqueue***	Specify the MQ queue name.
-q ***mqqueuemanager***	Specify the MQ queue manager name.
-b ***blocksize***	Specify the blocksize. The default value = 1.
-d ***dateformat***	Specify the date format. The default value = %Y-%m-%d %H:%M:%S
-e	Specify that events are transactional.
–g ***gdconfig***	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y ***logconfigfile***	Specify the log configuration file.

Using the Tervela Data Fabric Adapter

The Tervela adapter supports publish and subscribe operations on a hardware-based or software-based Tervela fabric. You must install the Tervela run-time libraries to use the adapter.

Subscriber usage:

dfesp_tva_adapter -k sub -h *url* —u *tvauserid*
—p *tvapassword* —t *tvaprimarytmx* —f *tvatopic*
—c *tvaclientname* -m *tvamaxoutstand* —b *numbufferedmsgs*
-o *urlhostport* <-s *tvasecondarytmx*> <-l *tvalogfile*>
<-w *tvapubbwlimit*> <-r *tvapubrate*><-e *tvapubmsgexp*>
<-g *gdconfig*> <-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*>

Publisher usage:

dfesp_tva_adapter -k pub -h *url* —u *tvauserid*
—p *tvapassword* —t *tvaprimarytmx* —f *tvatopic*
—c *tvaclientname* -n *tvasubname*
-o *urlhostport* <-s *tvasecondarytmx*> <-l *tvalogfile*>
<-g *gdconfig*> <-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
-h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***”. Append the following for subscribers: **?snapshot=true** | **false** Append the following for subscribers if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
-u *tvauserid*	Specify the Tervela user name.
-p *tvapassword*	Specify the Tervela password.
-t *tvaprimarytmx*	Specify the Tervela primary TMX.
-f *tvatopic*	Specify the Tervela topic.
-c *tvaclientname*	Specify the Tervela client name.
-m *tvamaxoutstand*	Specify the Tervela maximum number of unacknowledged messages.
-b *numbufferedmsgs*	Specify the maximum number of messages buffered by a standby subscriber connector.
-o *urlhostport*	Specify the *host*:*port* string sent in connector metadata message
—s *tvasecondarytmx*	Specify the Tervela secondary TMX.
—i*tvalogfile*	Specify the Tervela log file. The default is syslog.

Parameter	Definition
–w ***tvapubbwlimit***	Specify the Tervela maximum bandwidth of published data (Mbps). The default value is 100.
-r ***tvapubrate***	Specify the Tervela publish rate (Kbps). The default value is 30.
–e ***tvapubmsgexp***	Tervela maximum time to cache published messages (seconds); the default value is 1
–g ***gdconfig***	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y ***logconfigfile***	Specify the log configuration file.

Using the Solace Systems Adapter

The Solace Systems adapter supports publish and subscribe operations on a hardware-based Solace fabric. You must install the Solace run-time libraries to use the adapter.

Subscriber usage:

dfesp_sol_adapter -k sub -h *url* -u *soluserid*
—p *solpassword* —v *solvpn* —t *soltopic*
—o *urlhostport* -n *numbufferedmsgs* <-b > <-g *gdconfig*>
<-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-f *protofile*> <-m *protomsg*>

Publisher usage:

dfesp_sol_adapter -k pub -h *url* -u *soluserid*
—p *solpassword* —v *solvpn* —t *soltopic*
—o *urlhostport* <-b > <-q *buspersistencequeue*><-g *gdconfig*>
<-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-f *protofile*> <-m *protomsg*>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use

Parameter	Definition
-h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***". Append the following for subscribers: **?snapshot= true | false**. Append the following for subscribers if needed: • **?collapse= true | false** • **?rmretdel=true | false**
-u *soluserid*	Specify the Solace user name.
-p *solpassword*	Specify the Solace password.
-s *solhostport*	Specify the Solace *host*:*port*.
-v *solvpn*	Specify the Solace VPN name.
-t *soltopic*	Specify the Solace topic.
-o *urlhostport*	Specify the *host*:*port* field in the metadata topic subscribed to by the connector.
–n *numbufferedmsgs*	Specify the maximum number of messages buffered by a standby subscriber connector.
–g *gdconfig*	Specify the guaranteed delivery configuration file.
–l native | solace | tervela	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | debug | info | warn | error | fatal | off	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y *logconfigfile*	Specify the log configuration file.
-b	Use Solace Guaranteed Messaging. By default, Solace Direct Messaging is used.
–q *buspersistencequeue*	Specify the queue name used by Solace Guaranteed Messaging publisher.
-f *protofile*	Specify the .proto file that contains the message used for Google Protocol buffer support.
-m *protomsg*	Specify the message itself in the .proto file that is specified by the **protofile** parameter.

Using the Tibco Rendezvous (RV) Adapter

The Tibco RV adapter supports publish and subscribe operations using a Tibco RV daemon. You must install the Tibco RV run-time libraries to use the adapter.

Subscriber usage:

dfesp_tibrv_adapter -k sub -h *url* -f *tibrvsubject* —t *tibrvtype*
-s *tibrvservice*<-n *tibrvnetwork*> <-m *tibrvdaemon*>
<-d *dateformat*> <-g *gdconfig*>
<-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*>

Publisher usage:

dfesp_tibrv_adapter -k pub -h *url* -f *tibrvsubject* —t *tibrvtype*
-s *tibrvservice*<-n *tibrvnetwork*> <-m *tibrvdaemon*><-b *blocksize*>
<-d *dateformat*> <-e><-g *gdconfig*>
<-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*>

Table 12.1 *Parameter Definitions*

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use.
-h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host:port/project/continuousquery/window*"**.**** Append the following for subscribers: **"?snapshot= true | false "**. Append the following for subscribers if needed: • **?collapse= true | false** • **?rmretdel= true | false**
-f *tibrvsubject*	Specify the Tibco RV subject.
-t *tibrvtype*	Specify “binary” or “CSV”.
-s *tibrvservice*	Specify the Tibco RV service.
-n *tibrvnetwork*	Specify the Tibco RV network.
-m *tibrvdaemon*	Specify the Tibco RV daemon.
-b *blocksize*	Specify the block size. The default value is 1.
—d *dateformat*	Specify the date format. The default is %Y-%m-%d %H:%M: %S.

Parameter	Definition
–e	Specify that events are transactional.
–g ***gdconfig***	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y ***logconfigfile***	Specify the log configuration file.

Using the PI Adapter

The PI Adapter supports publish and subscribe operations against a PI Asset Framework (PI) server. You must install the PI AF Client from OSISoft in order to use the adapter.

Subscriber usage:

dfesp_pi_adapter -k sub -h *url* -s *afelement*<-t> <-p *pisystem*><-d *afdatabase*> <-r *afrootelement*><-a *afattribute*><-g *gdconfig*><-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal> <-y *logconfigfile*>

Publisher usage:

dfesp_pi_adapter -k pub-h *url* -s *afelement*<-t> <-p *pisystem*><-d *afdatabase*><-r *afrootelement*> <-a *afattribute*><-c><-b *blocksize*> <-e><-g *gdconfig*> <-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal> <-y *logconfigfile*>

Parameter	Definition
-k	Specify **sub** for subscriber use and **pub** for publisher use
-h ***url***	Specify the dfESP publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window***. Append the following for subscribers: **?snapshot=true** | **false**. Append the following for subscribers if needed: **?collapse=true** | **false**.
-s ***afelement***	Specify the AF element or element template name. Wildcards are supported.

Parameter	Definition
-t	Specify that *afelement* is the name of an element template.
-p *pisystem*	Specify the PI system.
–d *afdatabase*	Specify the AF database
–r *afrootelement*	Specify a root element in the AF hierarchy from which to search for *afelement*.
-a *afattribute*	Specify an attribute in *afelement* and ignore other attributes there.
-b *blocksize*	Specify the block size. The default value is 1.
-c *archivetimestamp*	Specify that when connecting to the AF server, retrieve all archived values from the specified timestamp onwards.
—e	Specify that events are transactional.
–g *gdconfig*	Specify the guaranteed delivery configuration file.
–l native | solace | tervela	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
-j trace | debug | info | warn | error | fatal | off	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y *logconfigfile*	Specify the log configuration file.

Using the Teradata Subscriber Adapter

The Teradata adapter supports subscribe operations against a Teradata server, using the Teradata Parallel Transporter for improved performance. You must install the Teradata Tools and Utilities (TTU) to use the adapter.

Note: A separate database adapter uses generic DataDirect ODBC drivers for the Teradata platform. For more information, see "Using the Database Adapter" on page 165.

Usage:

dfesp_tdata_adapter -h *url* -d stream | update | load -u *username* -x *userpwd* -t *tablename* -s *servername* -a *maxsessions* -n *minsessions* -i true | false <-b *batchperiod*> <-q *stage1table*> <-r *stage2table*> <-c *connectstring*> <-z *tracelevel*> <-g *gdconfig*> <-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*>

Parameter	Description
-h *url*	Specify the dfESP publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window*?snapshot=true | false**". Append **?collapse=true | false** for subscribers if needed. Append **?rmretdel=true | false** for subscribers if needed.
-d stream | update | load	Specify the Teradata operator. For more information about these operators, see the documentation provided with the Teradata Tools and Utilities.
-d *username*	Specify the Teradata user name.
-x *userpwd*	Specify the Teradata user password.
-t *tablename*	Specify the name of the target table on the Teradata server.
-s *servername*	Specify the name of the target Teradata server.
-a *maxsessions*	Specify the maximum number of sessions on the Teradata server. A Teradata session is a logical connection between an application and Teradata Database. It allows an application to send requests to and receive responses from Teradata Database. A session is established when Teradata Database accepts the user name and password of a user.
-n *minsessions*	Specify the minimum number of sessions on the Teradata server.
-n	Specify that the subscribed window contains insert-only data.
-b *batchperiod*	Specify the batch period in seconds. This parameter is required when **-d update**.
-q *stage1table*	Specify the name of the first staging table on the Teradata server. This parameter is required when **-d load**.
-r *stage2table*	Specify the name of the second staging table on the Teradata server. This parameter is required when **-d load**.
-c *connectstring*	Specify the connect string to be used by the ODBC driver to access the staging and target tables on the Teradata server. Use the form **"DSN=*dsnmane*;uid=*userid*;pwd=*password***. This parameter is required when **-d load**.

Parameter	Description
-z ***tracelevel***	Specify the trace level for Teradata messages written to the trace file in the current working directory. The trace file is named *operator*1.txt. Default value is 1 (TD_OFF). Other valid values are: 2 (TD_OPER), 3 (TD_OPER_CLI), 4 (TD_OPER_OPCOMMON), 5 (TD_OPER_SPECIAL), 6 (TD_OPER_ALL), 7 (TD_GENERAL), and 8 (TD_ROW).
-g ***gdconfig***	Specify the guaranteed delivery configuration file.
-l native | **solace** | **tervela**	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default native transport, use the required client configuration files specified in the description of the **C++ C_dfESPpubsubSetPubsubLib()** API call.
-j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. Use the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call or in the engine **initialize()** call.
-y ***logconfigfile***	Specify the log configuration file.

Chapter 13

Enabling Guaranteed Delivery

Overview to Guaranteed Delivery

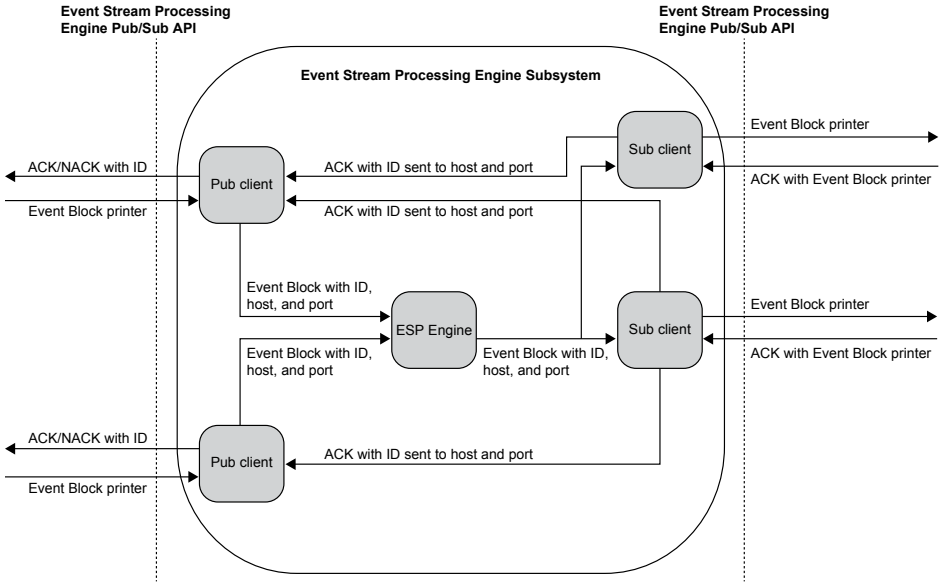
Both the Java and C publish and subscribe (pub/sub) APIs support guaranteed delivery between a single publisher and multiple subscribers. Guaranteed delivery assumes a model where each event block that is published into a source window generates exactly one event block in a subscribed window. This one block in, one block out principle must hold for all published event blocks. The guaranteed delivery acknowledgment mechanism has no visibility into the event processing performed by the model.

When a publish or subscribe connection is started, a client is established to perform various publish/subscribe activities. When a publish connection is started, the number of guaranteed subscribers required to acknowledge delivery of its event blocks is specified. The time-out value used to generate negative acknowledgments upon non-receipt from all expected subscribers is also specified. Every event block injected by the publisher contains a unique 64-bit ID set by the publisher. This ID is passed back to the publisher from the publish client with every acknowledgment and negative acknowledgment in a publisher user-defined callback function. The function is registered when the publish client is started.

When a subscribe connection is started, the subscribe client is passed a set of guaranteed delivery publishers as a list of host and port entries. The client then establishes an acknowledged connection to each publisher on the list. The subscriber calls a new publish/subscribe API function to trigger an acknowledgment.

Event blocks contain new host, port, and ID fields. All event blocks are uniquely identified by the combination of these fields, which enables subscribers to identify duplicate (that is, resent) event blocks.

Display 13.1 *Guaranteed Delivery Data Flow Diagram*



Note:

Please note the following:

- Publishers and subscribers that do not use the guaranteed-delivery-enabled API functions are implicitly guaranteed delivery disabled.
- Guaranteed delivery subscribers can be mixed with non-guaranteed delivery subscribers.
- A guaranteed delivery-enabled publisher might wait to begin publishing until a READY callback has been received. This indicates that its configured number of subscribers have established their acknowledged connections back to the publisher.
- Event blocks received by a guaranteed-delivery-enabled subscriber as a result of a snapshot generated by the SAS Event Stream Processing Engine are not acknowledged.
- Under certain conditions, subscribers receive duplicate event blocks. These conditions include the following:
- A publisher begins publishing before all related subscribers have started. Any started subscriber can receive duplicate event blocks until the number of started subscribers reaches the number of required acknowledgments passed by the publisher.
- A guaranteed delivery-enabled subscriber disconnects while the publisher is publishing. This triggers the same scenario described previously.
- A slow subscriber causes event blocks to time-out, which triggers a not acknowledged to the publisher. In this case all subscribers related to the publisher receives any resent event blocks, including those that have already called **`C_dfESPGDsubscriberAck()`** for those blocks.

- If a guaranteed delivery-enabled subscriber fails to establish its acknowledged connection, it retries at a configurable rate up to a configurable maximum number of retries.
- Suppose that a guaranteed delivery-enabled publisher injects an event block that contains an ID, and that the ID is present in the publish client's not acknowledged-ID list. In that case, the inject call is rejected by the publish client. The ID is cleared from the list when the publish client passes it to the ACK/NACK callback function of the new publisher.

Guaranteed Delivery Success Scenario

In the context of guaranteed delivery, the publisher and subscriber are customer applications that are the endpoints in the data flow. The subscribe and publish clients are event stream processing code that implements the publish/subscribe API calls made by the publisher and subscriber.

The flow of a guaranteed delivery success scenario is as follows:

1. The publisher passes an event block to the publish client, where the ID field in the event block has been set by the publisher. The publish client fills in the host-port field, adds the ID to its unacknowledged ID list, and injects it to the SAS Event Stream Processing Engine.
2. The event block is processed by the SAS Event Stream Processing Engine and the resulting Inserts, Updates, or Deletes on subscribe windows are forwarded to all subscribe clients.
3. A guaranteed delivery-enabled subscribe client receives an event block and passes it to the subscriber by using the standard subscriber callback.
4. Upon completion of all processing, the subscribers call a new API function with the event block pointer to trigger an acknowledgment.
5. The subscribe client sends the event block ID on the guaranteed delivery acknowledged connection that matches the host or port in the event block, completely bypassing the SAS Event Stream Processing Engine.
6. Upon receipt of the acknowledgment, the publish client increments the number of acknowledgments received for this event block. If that number has reached the threshold passed to the publish client at start-up, the publish client invokes the new guaranteed delivery callback with parameters acknowledged and ID. It removes the ID from the list of unacknowledged IDs.

Guaranteed Delivery Failure Scenarios

There are three failure scenarios for guaranteed delivery flows:

Scenario	Description
Event Block Time-out	• An event block-specific timer expires on a guaranteed-delivery-enabled publish client, and the number of acknowledgments received for this event block is below the required threshold. • The publish client invokes the new guaranteed delivery callback with parameters **`NACK`** and **`ID`**. No further retransmission or other attempted recovery by the SAS Event Stream Processing Engine publish client or subscribe client is undertaken for this event block. The publisher most likely backs out this event block and resends. • The publish client removes the ID from the list of unacknowledged IDs.
Invalid Guaranteed Delivery Acknowledged Connect Attempt	• A guaranteed-delivery-enabled publish client receives a connect attempt on its guaranteed delivery acknowledged server but the number of required client connections has already been met. • The publish client refuses the connection and logs an error message. • For any subsequent event blocks received by the guaranteed delivery-enabled subscribe client, an error message is logged.
Invalid Event Block ID	• A guaranteed-delivery-enabled publisher injects an event block that contains an ID already present in the publish client's unacknowledged ID list. • The inject call is rejected by the publish client and an error message is logged.

Additions to the Publish/Subscribe API for Guaranteed Delivery

The publish/subscribe API provides the following methods to implement guaranteed delivery sessions:

- **`C_dfESPGDpublisherStart()`**
- **`C_dfESPGDsubscriberStart()`**
- **`C_dfESPGDsubscriberAck()`**
- **`C_dfESPGDpublisherCB_func()`**
- **`C_dfESPGDpublisherGetID()`**

For more information, see "Functions for the C Publish/Subscribe API" on page 113. For publish/subscribe operations without a guaranteed delivery version of the function, call the standard publish/subscribe API function.

Configuration File Contents

The publish client and subscribe client reads a configuration file at start-up to get customer-specific configuration information for guaranteed delivery. The format of both of these files is as follows.

Guaranteed Delivery-enabled Publisher Configuration File Contents	
Local port number for guaranteed delivery acknowledged connection server.	
Time-out value for generating not acknowledged, in seconds.	
Number of received acknowledgments required within time-out period to generate acknowledged instead of not acknowledged.	
File format:	**GDpub_port=<port> GDpub_timeout=<timeout> GDpub_numSubs=<number of subscribers generating acknowledged>**

Guaranteed Delivery-enabled Subscriber Configuration File Contents	
List of guaranteed delivery-enabled publisher host or port entries. Each entry contains a host:port pair corresponding to a guaranteed delivery-enabled publisher from which the subscriber wishes to receive guaranteed delivery event blocks.	
Acknowledged connection retry interval, in seconds.	
Acknowledged connection maximum number of retry attempts.	
File Format:	**GDsub_pub=<host:port> … … GDsub_retryInt=<interval> GDsub_maxretries=<max>**

Publish/Subscribe API Implementation of Guaranteed Delivery

Here is an implementation of the C publish/subscribe API for publishing. The Java implementation is similar.

```
/**
```

```
* Type enumeration for Guaranteed Delivery event block status.
*/
typedef enum {
ESP_GD_READY,
ESP_GD_ACK,
ESP_GD_NACK
} C_dfESPGDStatus;
/**
* The publisher client callback function to be called for notification of
* Guaranteed Delivery (GD) event block status.
* @param eventBlockStatus
*    ESP_GD_READY: required number of subscriber ACK connections
*                  are established
*    ESP_GD_ACK:   ACKs have been received from all required subscribers
*                  for this event block ID
*    ESP_GD_NACK:  an event block has timed out and insufficient ACKS
*                  have been received for the event block ID
* @param eventBlockID the event block ID (0 if status is ESP_GD_READY)
* @param ctx the user context pointer passed to C_dfESPGDpublisherStart()
*/
typedef void (*C_dfESPGDpublisherCB_func)
             (C_dfESPGDStatus eventBlockStatus, int64_t eventBlockID, void *ctx);

* The guaranteed delivery version of C_dfESPpublisherStart().
* @param serverURL string with format = "dfESP://<hostname>:<port>/
     <project name>/<continuous query name>/<window name>"
* @param errorCallbackFunction function pointer to user-defined error
     catching function, NULL for no error callback.
* @param ctx a user-defined context pointer.
* @param configFile the guaranteed delivery configuration file for this
*    publisher
* @param gdCallbackFunction function pointer to user-defined guaranteed
     delivery callback function
* @return pointer to the created object, NULL if failure.
*/
DFESPPS_API clientObjPtr C_dfESPGDpublisherStart(char *serverURL,
C_dfESPpubsubErrorCB_func errorCallbackFunction,
void *ctx, char *configFile,
C_dfESPGDpublisherCB_func gdCallbackFunction);
/**
* Return sequentially unique IDs to a guaranteed-delivery-enabled publisher
* to be written to event blocks before injecting them to a guaranteed
* delivery-enabled publish client.
* @return event ID
*/
DFESPPS_API C_ESP_int64_t C_dfESPGDpublisherGetID();
```

Here is a C publish/subscribe implementation for subscribing. The Java implementation is similar.

```
/**
* The guaranteed delivery version of C_dfESPsubscriberStart().
* @param serverURL string with format = "dfESP://<hostname>:<port>/
     <project name>/<continuous query name>/<window name>
     ?snapshot={true|false}[?collapse={true|false}]"
* @param callbackFunction function pointer to user-defined subscriber
     callback function
```

```
* @param errorCallbackFunction function pointer to user-defined error
     catching function, NULL for no error callback
* @param ctx a user defined context pointer
* @param configFile the GD configuration file for this subscriber
* @return pointer to the created object, NULL if failure
*/
DFESPPS_API clientObjPtr C_dfESPGDsubscriberStart(char *serverURL,
C_dfESPsubscriberCB_func callbackFunction,
C_dfESPpubsubErrorCB_func errorCallbackFunction,
void *ctx, char *configFile);
/**
* Trigger a guaranteed delivery acknowledgement of an event block.
* @param eventBlock the event block pointer
* @return int 1 = success, 0 = failure
*/
DFESPPS_API int C_dfESPGDsubscriberAck(clientObjPtr client,
                                       C_dfESPeventblock) ;
```

Chapter 14

Implementing 1+N-Way Failover

Overview to 1+N-Way Failover

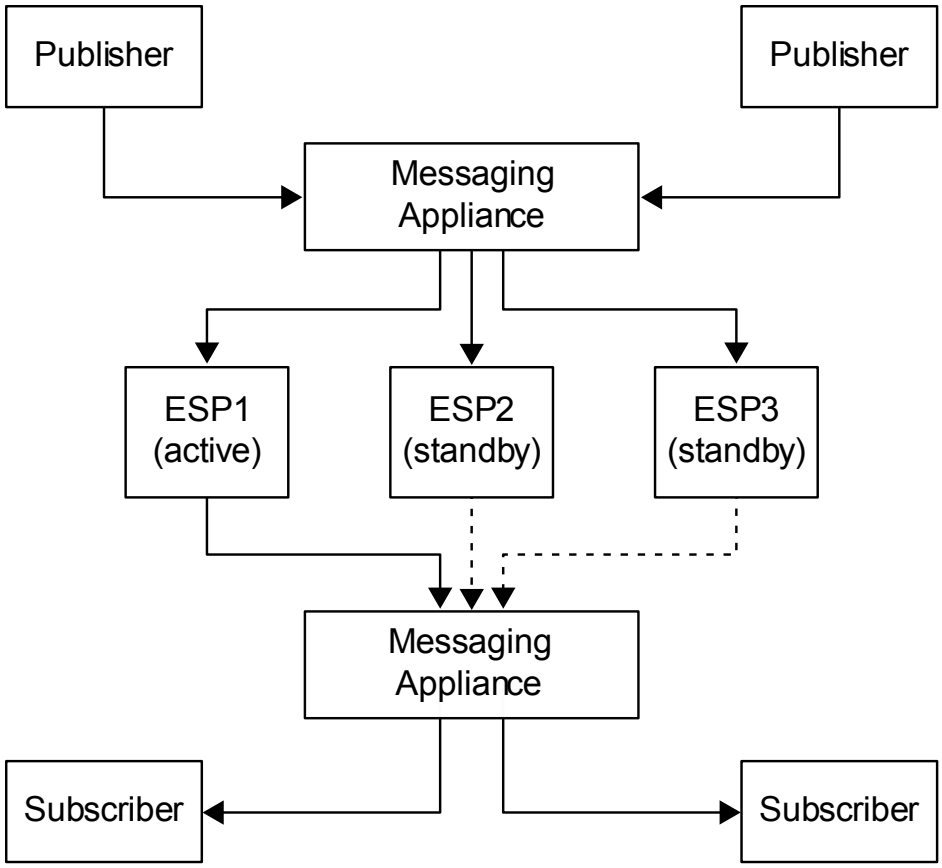
SAS Event Stream Processing Engine can use third-party messaging appliances to provide 1+N-Way Failover. Event stream processing publishers and subscribers work with packages of events called event blocks when they interface with the engine. When traversing a messaging appliance, event blocks are mapped one-to-one to appliance messages. Each payload message contains exactly one event block. These event blocks contain binary event stream processing data. A payload appliance message encapsulates the event block and transports it unmodified.

The sections that follow use the terms “message” and “event block” interchangeably. They also use the following terms interchangeably: messaging appliance, appliance, message fabric, and message bus. The term active/standby identifies the state of any

event stream processors in a 1+N cluster of event stream processors. The term primary/secondary identifies the state of an appliance with respect to another appliance in a redundant pair. The terms 1+N, failover, cluster, and combinations of these terms are used interchangeably.

The following diagram shows how SAS Event Stream Processing engine integrates with third-party messaging appliances to provide failover. It shows two separate messaging appliances, one between publishers and engines (ESPs) and a second between ESPs and subscribers. In actual deployments, these do not have to be separate appliances. Regardless of whether publishers and subscribers use the same or different appliances, there are two messaging appliances for each virtual messaging appliance — a primary and secondary for messaging appliance failover.

Figure 14.1 *Engine Integration with Third-Party Messaging Appliances*



In this diagram, ESP1 is the active engine (on start-up at least). ESP2 and ESP3 are standbys that are receiving published event blocks. They do not send processed event blocks to the subscriber messaging appliance. This distinction is depicted with dotted arrows. The event stream processing messaging appliance connector for subscribe services is connected to the fabric. It does not actually sending event blocks to the fabric until one of them becomes the active on failover.

All ESPs in a 1+N failover cluster must implement the same model, because they are redundant. It is especially important that all ESPs in the cluster use the same engine name. This is because the engine name is used to coordinate the topic names on which messages are exchanged through the fabric.

Publishers and subscribers can continue to use the ESP API even when they are subscribing or publishing through the messaging appliance for failover.

The following transport options are supported so that failover can be introduced to an existing implementation without reengineering the subscribers and publishers:

- native
- Tervela
- Solace

However, when you use the messaging appliance for publish/subscribe, the event stream processing API uses the messaging appliance API to communicate with the messaging appliance. It does not establish a direct TCP connection to the event stream processing publish/subscribe server.

Engines implement Tervela or Solace connectors to communicate with the messaging appliance. Like client publishers and subscribers, they are effectively subscribers and publishers. They subscribe to the messaging appliance for messages from the publishers. They publish to the message appliance so that it can publish messages to the subscribers.

These fabrics support using direct (that is, non-persistent) or persistent messaging modes. For this application, the Tervela connector requires that Tervela fabrics use persistent messaging for all publish/subscribe communication between publishers, ESPs, and subscribers. Solace fabrics can use either direct or persistent messaging. Enabling persistent messaging on the appliance implies the following:

- The message bus guarantees delivery of messages to and from its clients using its proprietary acknowledgment mechanisms. Duplicate message detection, lost message detection, retransmissions, and lost ACK handling are handled by the messaging bus.
- Upon re-connection of any client and its re-subscription to an existing topic, the message bus replays all the messages that it has persisted for that topic. The number of messages or time span that this covers is appliance-configuration dependent.
- At the start of the day, the appliance should be purged of all messages on related topics. Message IDs must be synchronized across all connectors.

The ESPs are deployed in a 1+N redundant manner. This means the following:

- All the ESPs in the 1+N cluster receive messages from the publishers.
- Only the active ESP in the 1+N cluster publishes messages to the subscribers.
- One or more backup ESPs in a 1+N cluster might be located in a remote data center, and connected over the WAN.

For simplicity, the reference architecture diagram illustrates one cluster of 1+N redundant ESPs. However, there can be multiple clusters of ESPs, each subscribing and publishing on a different set of topics. A single publisher can send messages to multiple clusters of ESPs. A single subscriber can receive messages from multiple ESPs.

The message bus provides a mechanism to signal to an ESP that it is the active ESP in the cluster. The message bus provides a way for an ESP, when notified that it is active, to determine the last message published by the previously active ESP. The newly active ESP can resume publishing at the appropriate point in the message stream.

Sequence numbering of messages is managed by the ESP's appliance connectors for the following purposes:

- detecting duplicates
- detecting gaps
- determining where to resume sending from after an ESP fail-over

An ESP that is brought online resynchronizes with the day's published data and the active ESP. The process occurs after a failure or when a new ESP is added to a 1+N cluster.

ESPs are deployed in 1+N redundancy clusters. All ESPs in the cluster subscribe to the same topics on the message bus, and hence receive exactly the same data. However, only one of the ESPs in the cluster is deemed the active ESP at any time. Only the active ESP publishes data to the downstream subscribers.

Note: Solace functionality is not available on HP Itanium or AIX platforms.

Note: Tervela functionality is not available on HP Itanium, AIX, or SPARC platforms.

Topic Naming

Overview to Topic Naming

Topic names are mapped directly to engine (ESP) windows that send or receive event blocks through the fabric. Because all ESPs in a 1+N cluster implement the same model, they also use an identical set of topics on the fabric. However, to isolate publish flows from subscribe flows to the same window, all topic names are appended with an "in" or "out" designator. This enables clients and ESP appliance connectors to use appliance subscriptions and publications, where event blocks can flow only in one direction.

Current client applications continue to use the standard ESP URL format, which includes a *host*:*port* section. No publish/subscribe server exists, so *host*:*port* is not interpreted literally. It is overloaded to indicate the target 1+N cluster of ESPs. All of these ESPs have the same engine name, so a direct mapping between *host*:*port* and engine name is established to associate a set of clients with a specific 1+N ESP cluster.

You create this mapping by configuring each ESP appliance connector with a "**urlhostport**" parameter that contains the *host*:*port* section of the URL passed by the client to the publish/subscribe API. This parameter must be identical for all appliance connectors in the same 1+N failover cluster.

Solace

The topic name format used on Solace appliances is as follows: ***host:port/project/contquery/window/direction***, where *direction* takes the value "I" or "O". Because all this information is present in a client URL, it is easy for clients to determine the correct appliance topic. ESP appliance connectors use their configured "**urlhostport**" parameter to derive the "*host*:*port*" section of the topic name, and the rest of the information is known by the connector.

Tervela

The topic name format used on Tervela appliances is as follows: "**SAS.ENGINES.*engine.project.contquery.window.direction***", where *direction* takes the value "IN" or "OUT". ESP appliance connectors know this information, so it is easy for them to determine the correct appliance topic.

Clients must be able to map the "*host*:*port*" section of the received URL to the engine section of the topic name. This mapping is obtained by the client by subscribing to a special topic named **SAS.META.*host*:*port***.. The ESP appliance connectors use their

configured "**`urlhostport`**" parameter to build this topic name,. They publish a metadata message to the topic that includes the "*host*:*port*" to engine mapping. Only after receiving this message can clients send or receive event block data. ESP appliance connectors automatically send this message when the ESP model is started.

Failover Mechanisms

Overview to Failover Mechanisms

If the active engine (ESP) in a failover cluster fails, the standby ESP appliance connectors are notified. Then one of them becomes the new active ESP. The fabric tells the new active connector the ID of the last message that it received on the window-specific "out" topic. The new active connector begins sending data on that "out" topic with ID + 1.

When appliance connectors are inactive, they buffer outbound messages (up to a configurable maximum) so that they can find messages starting with ID+1 in the buffer if necessary.

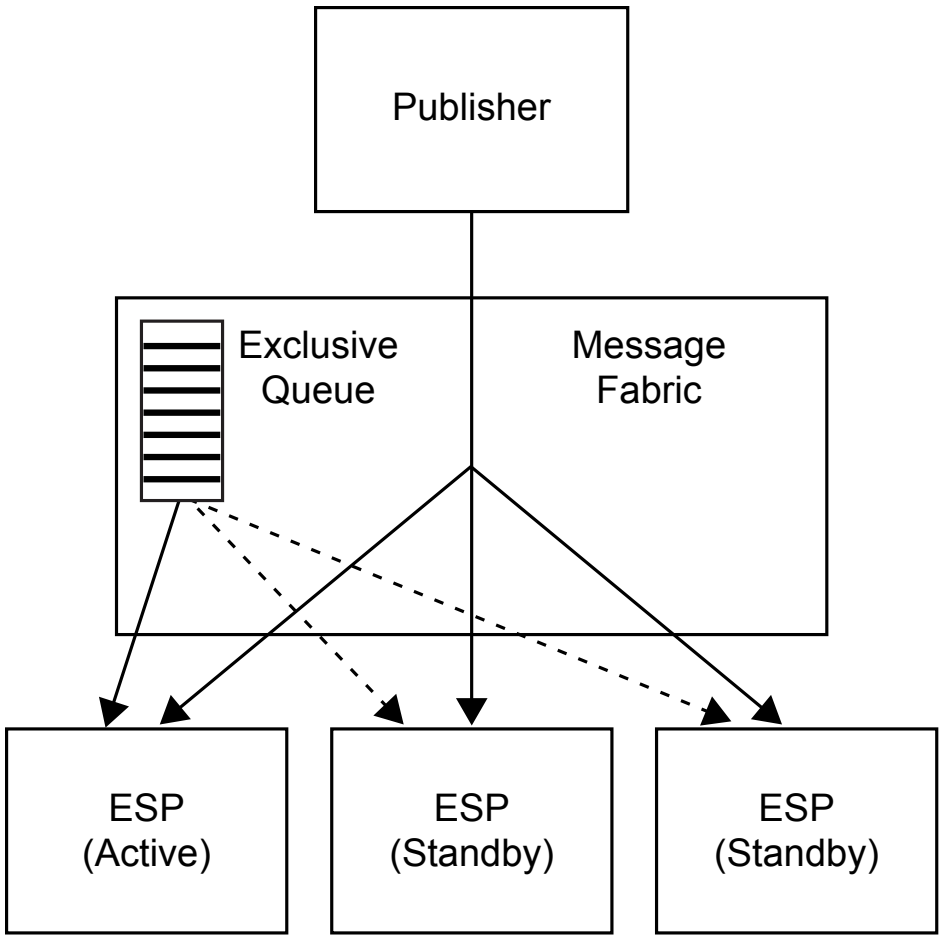
Note: Failover support is unavailable when Google Protocol buffer support is enabled.

Determining ESP Active/Standby State (Solace)

For Solace appliances, an exclusive messaging queue is shared amongst all the engines (ESPs) in the 1+N cluster. The queue is used to signal active state. No data is published

to this queue. It is used as a semaphore to determine which ESP is the active at any point in time.

Figure 14.2 *Determining Active State*



ESP active/standby status is coordinated among the engines using the following mechanism:

1. When an ESP subscriber appliance connector starts, it tries, as a queue consumer, to bind to the exclusive queue that has been created for the ESP cluster.
2. If the connector is the first to bind to the queue, it receives a “Flow Active” indication from the messaging appliance API. This signals to the connector that it is now the active ESP.
3. As other connectors bind to the queue, they receive a “Flow Inactive” indication. This indicates that they are standby ESPs, and should not be publishing data onto the message bus.
4. If the active ESP fails or disconnects from the appliance, one of the standby connectors receives a “Flow Active” indication from the messaging appliance API. Originally, this is the second standby connector to connect to the appliance. This indicates that it is now the active ESP in the cluster.

Determining ESP Active/Standby State (Tervela)

When using the Tervela Data Fabric, ESP active/standby status is signaled to the ESPs using the following mechanism:

1. When an ESP subscriber appliance connector starts, it attempts to create a "well-known" Tervela inbox. It uses the engine name for the inbox name, which makes it specific to the failover cluster. If successful, that connector takes ownership of a system-wide Tervela GD context, and becomes active. If the inbox already exists, another connector is already active. The connector becomes standby and does not publish data onto the message bus.
2. When a connector becomes standby, it also connects to the inbox, and sends an empty message to it.
3. The active connector receives an empty message from all standby connectors. It assigns the first responder the role of the active standby connector by responding to the empty message. The active connector maintains a map of all standby connectors and their status.
4. If the active connector receives notification of an inbox disconnect by a standby connector, it notifies another standby connector to become the active standby, using the same mechanism.
5. If the active ESP fails, the inbox also fails. At this point the fabric sends a `TVA_ERR_INBOX_COMM_LOST` message sent to the connected standby connectors.
6. When the active standby connector receives a `TVA_ERR_INBOX_COMM_LOST` message, it becomes the active ESP in the failover cluster. It then creates a new inbox as described in step 1.
7. When another standby connector receives a `TVA_ERR_COMM_LOST` message, it retains standby status. It also finds the new inbox, connects to it, and send an empty message to it.

New ESP Active Actions on Failover (Solace)

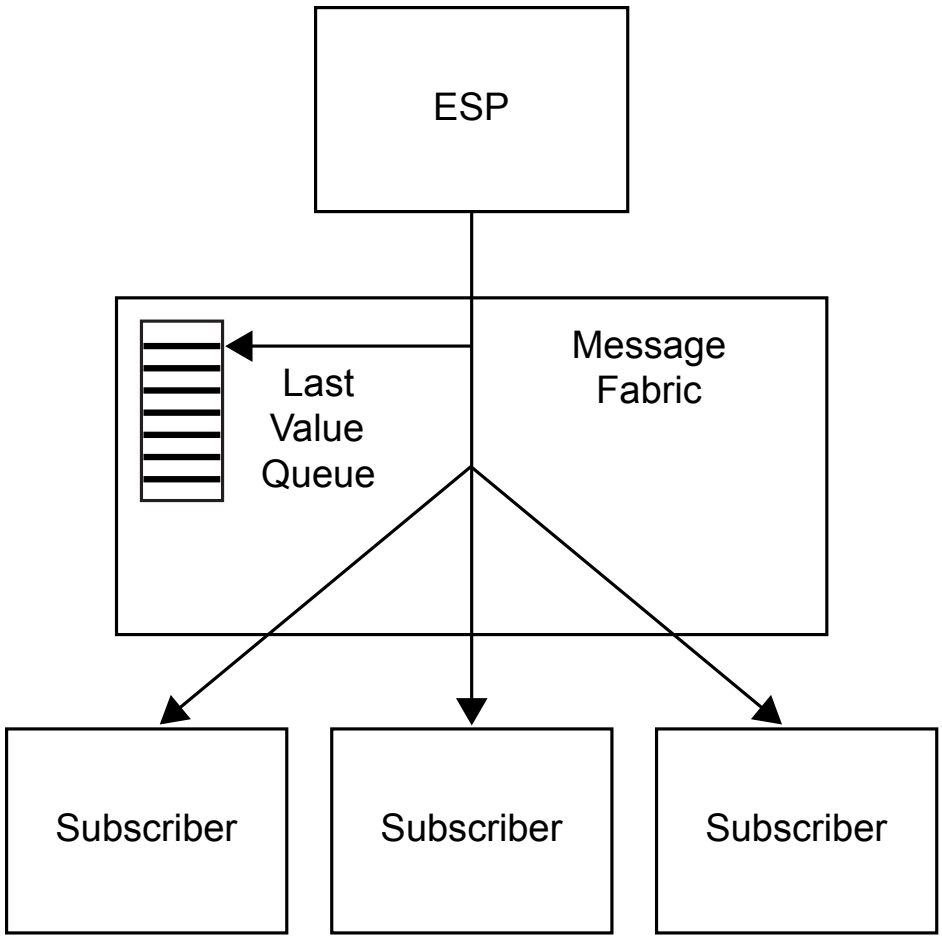
The newly active engine (ESP) determines, from the message bus, the last message published by the previously active ESP for the relevant window. To assist in this process, guaranteed messaging Last Value Queues (LVQs) are used.

LVQs are subscribed to the same "out" topics that are used by the appliance connectors. An LVQ has the unique characteristic that it maintains a queue depth of one message, which contains the last message published on the topic to which it subscribed. When the ESP can publish messages as "direct" or "guaranteed", those messages can always be received by a guaranteed messaging queue that has subscribed to the message topic. Thus, the LVQ always contains the last message that an ESP in the cluster published onto the message bus.

When an ESP receives a "Flow Active" indication, it binds to the LVQ as a browser. It then retrieves the last message published from the queue, saves its message ID, disconnects from the LVQ, and starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent messages from

the queue that it maintained while it was inactive. It can ignore newly received messages until the one with ID = saved message ID + 1 is received.

Figure 14.3 *Last Value Queues*



New ESP Active Actions on Failover (Tervela)

Active Tervela appliance connectors own a cluster-wide Tervela GD context with a name that matches the configured "**`tvaclientname`**" parameter. This parameter must be identical for all subscribe appliance connectors in the same failover cluster. When a connector goes active because of a failover, it takes over the existing GD context. This allows it to query the context for the ID of the last successfully published message, and this message ID is saved.

The connector then starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. Alternatively, it can ignore newly received messages until the one with ID = saved message ID + 1 is received.

Restoring Failed Active ESP State after Restart

When a failed active is manually brought back online, it is made available as a standby when another ESP in the cluster is currently active. If the appliance is operating in "direct" mode, persisted messages on the topic do not replay. This standby ESP remains out-of-sync with other ESPs in the failover cluster that have already had event blocks injected into that window. When the appliance is in "persistence" or "guaranteed" mode, it replays as much data as it has persisted on the "in" topic when a client reconnects. The amount of data that is persisted depends on appliance configuration and disk resources. In many cases, the data persisted might not be enough to cover up one day of messages.

Using ESP Persist/Restore

To guarantee that a rebooted engine (ESP) can be fully synchronized with other running ESPs in a failover cluster, use the ESP persist/restore feature with an appliance in "guaranteed" mode. This requires that ESP state is periodically persisted by any single ESP in the failover cluster. A persist can be triggered by the model itself, but in a failover cluster this generates redundant persist data.

Alternatively, a client can use the publish/subscribe API to trigger a persist by an ESP engine. The URL provided by the client specifies *host*:*port*, which maps to a specific ESP failover cluster. The messaging mechanism guarantees that only one ESP in the cluster receives the message and executes the persist. On Solace appliances, this is achieved by setting Deliver-To-One on the persist message to the metadata topic. On the Tervela Data Fabric this is achieved by sending the persist message to an inbox owned by only one ESP in the failover cluster.

The persist data is always written to disk. The target path for the persist data is specified in the client persist API method. Any client that requests persists of an ESP in a specific failover cluster should specify the same path. This path can point to shared disk, so successive persists do not have to be executed by the same ESP in the failover cluster.

The other requirement is that the model must execute a restore on boot so that a rebooted standby ESP can synchronize its state using the last persisted snapshot. On start-up, appliance connectors always get the message ID of the last event block that was restored. If the restore failed or was not requested, the connector gets 0. This message ID is compared to those of all messages received through replay by a persistence-enabled appliance. Any duplicate messages are ignored.

Message Sequence Numbers

The message IDs that are used to synchronize ESP failovers are generated by the ESP engine. They are inserted into an event block when that event block is injected into the model. This ID is a 32-bit integer that is unique within the scope of its project/query/window, and therefore unique for the connector. When redundant ESP engines receive identical input, this ID is guaranteed to be identical for an event block that is generated by different engines in a failover cluster.

The message IDs that are used to synchronize a rebooted ESP with already published event blocks are generated by the inject method of the Solace or Tervela publisher client API. They are inserted into the event block when the event block is published into the appliance by the client. This ID is a 64-bit integer that is incremented for each event block published by the client.

Metadata Exchanges (Solace)

The Solace publish/subscribe API handles the **`C_dfESPpubsubQueryMeta()`** and **`C_dfESPpubsubPersistModel()`** methods as follows:

- The appliance connectors listen for metadata requests on a special topic named "urlhostport/M".
- The client sends formatted messages on this topic in request/reply fashion.
- The request messages are always sent using Deliver-To-One to ensure that no more than one ESP in the failover cluster handles the message.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

Metadata Exchanges (Tervela)

The Tervela publish/subscribe API handles the **`C_dfESPpubsubQueryMeta()`** method as follows:

- On start-up, appliance connectors publish complete metadata information about special topic "**`SAS.META.host:port`**". This information includes the "**`urlhostport`**" to engine mapping needed by the clients.
- On start-up, clients subscribe to this topic and save the received metadata and engine mapping. To process a subsequent **`C_dfESPpubsubQueryMeta()`** request, the client copies the requested information from the saved response(s).

The Tervela publish/subscribe API handles the **`C_dfESPpubsubPersistModel()`** method as follows.

- Using the same global inbox scheme described previously, the appliance connectors create a single cluster-wide inbox named "engine_meta".
- The client derives the inbox name using the received "**`urlhostport`**" - engine mapping, and sends formatted messages to this inbox in request/reply fashion.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

Required Software Components

Note the following requirements when you implement 1+N-way failover:

- The ESP model must implement the required Solace or Tervela publish and subscribe connectors. The subscribe connectors must have "**`hotfailover`**" configured to enable 1+N-way failover.
- Client publisher and subscriber applications must use the Solace or Tervela publish/subscribe API provided with SAS Event Stream Processing Engine. For C or C++ applications, the Solace or Tervela transport option is requested by calling **`C_dfESPpubsubSetPubsubLib()`** before calling **`C_dfESPpubsubInit()`**. For Java applications, the Solace or Tervela transport option is invoked by inserting dfx-esp-solace-api.jar or dfx-esp-tervela-api.jar into the classpath in front of dfx-esp-api.jar.
- The platforms hosting running instances of the connectors and clients must have the Solace or Tervela run-time libraries installed. This is because the Event Stream Processing Engine does not ship any appliance standard API libraries. The run-time environment must define the path to those libraries (using LD_LIBRARY_PATH on linux platforms, for example).

Required Client Configuration

A Solace client application requires a client configuration file named solace.cfg in the current directory to provide appliance connectivity parameters. A Tervela client application requires a client configuration file named client.config in the current directory to provide appliance connectivity parameters. See documentation of **`C_dfESPpubsubSetPubsubLib()`** publish/subscribe API function for details about the contents of these configuration files.

Required Appliance Configuration (Solace)

A Solace appliance used in this 1+N Way Failover topology requires this configuration at a minimum:

- A client user name and password to match the connector's **`soluserid`** and **`solpassword`** configuration parameters.
- A message VPN to match the connector's **`solvpn`** configuration parameter.
- On the message VPN, enable "Publish Subscription Event Messages".
- On the message VPN, enable "Client Commands" under "SEMP over Message Bus".
- On the message VPN, configure a nonzero "Maximum Spool Usage".
- If hot failover is enabled on subscriber connectors, create a single exclusive queue named "active_esp" in the message VPN. The subscriber connector that successfully binds to this queue becomes the active connector.
- If **`buspersistence`** is enabled, enable "Publish Client Event Messages" on the message VPN.
- If **`buspersistence`** is enabled, create exclusive queues for all clients subscribing to the Solace topics described below. The queue name must be equal to the topic name with "/*buspersistencequeue* appended. The *buspersistencequeue* is the queue configured on the publisher connector (for "/I" topics), or the queue configured on

the client subscriber (for "/O" topics). Add the corresponding topic to each configured queue.

- For high throughput deployments, modify the client profile to increase Queue Maximum Depth and Priority Queue Minimum Burst as needed.

Required Appliance Configuration (Tervela)

A Tervela appliance used in this 1+N Way Failover topology requires this configuration at a minimum:

- A client user name and password to match the connector's **`tvauserid`** and **`tvapassword`** configuration parameters.
- The inbound and outbound topic strings and associated schema. (See topic string formats described previously.)
- Publish or subscribe entitlement rights associated with a client user name described previously.

Chapter 15

Using Design Patterns

Overview to Design Patterns

Event stream processing models can be stateless, stateful, or mixed. The type of model that you choose affects how you design it. The combinations of windows that you use in your design pattern should enable fast and efficient event stream processing. One challenge when designing a mixed model is to identify sections that must be stateful and those that can be stateless, and then connecting them properly.

A stateless model is one where the indexes on all windows have the type **`pi_EMPTY`**. In this case, events are not retained in any window, and are essentially transformed and passed through. These models exhibit fast performance and use very little memory. They are well-suited to tasks where the inputs are inserts and when simple filtering, computation, text context analysis, or pattern matching are the only operations required.

A stateful model is one that uses windows with index types that store data, usually **`pi_RBTREE`** or **`pi_HASH`**. These models can fully process events with Insert, Update, or Delete opcodes. A stateful model facilitates complex relational operations such as joins and aggregations. Because events are retained in indexes, whenever all events are Inserts only, windows grow unbounded in memory. Thus, stateful models must process a mix of Inserts, Updates, and Deletes in order to remain bounded in memory.

The mix of opcodes can occur in one of two ways:

- The data source and input events have bounded key cardinality. That is, there are a fixed number of unique keys in the input stream (such as customer IDs). There can be many updates to these keys provided that the key cardinality is finite.
- A retention policy is enforced for the data flowing in, where the amount of data is limited by time or event count. The data is then automatically deleted from the system by the generation of internal retention delete events.

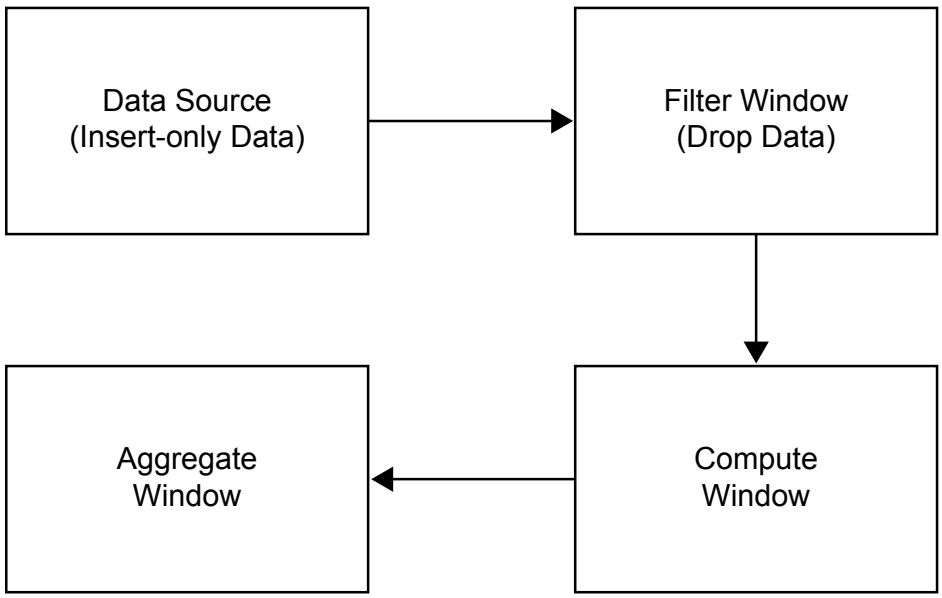
A mixed model has stateless and stateful parts. Often it is possible to separate the parts into a stateless front end and a stateful back end.

Design Pattern That Links a Stateless Model with a Stateful Model

To control memory growth in a mixed model, link the stateless and stateful parts with copy windows that enforce retention policies. Use this design pattern when you have insert-only data that can be pre-processed in a stateless way. Pre-process the data before you flow it into a section of the model that requires stateful processing (using joins, aggregations, or both).

For example, consider the following model:

Display 15.1 *Event Stream Processing Model with Insert-Only Data*



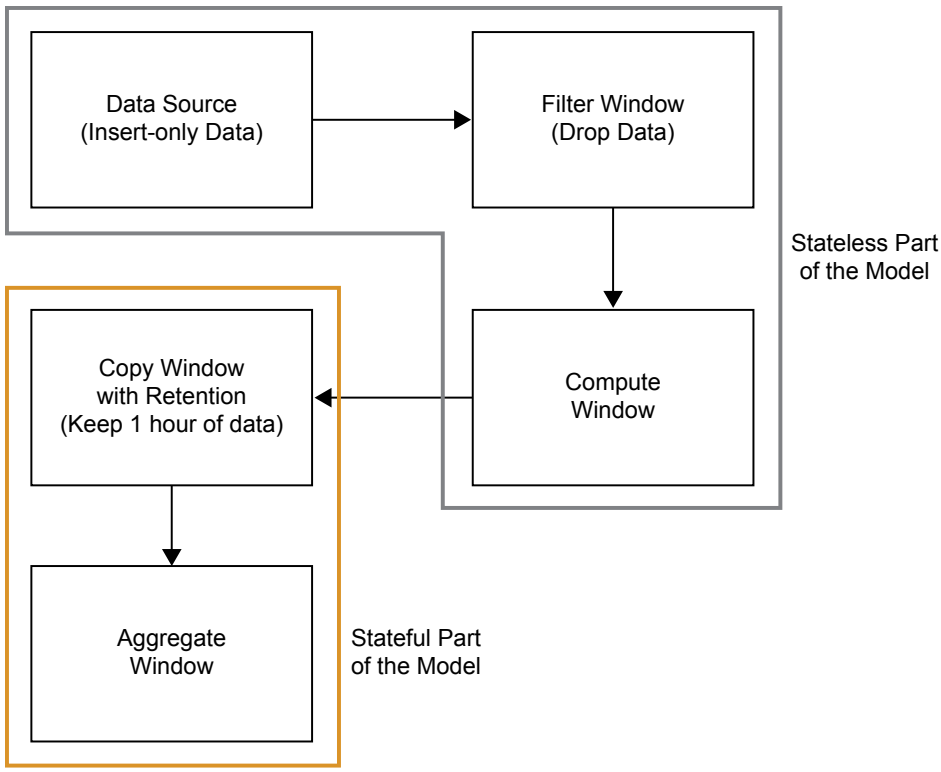
Here the data source is purely through Inserts. Therefore, the model can be made stateless by using an index type of **`pi_EMPTY`**. The filter receives inserts from the source, and drops some of them based on the filter criteria, so it produces a set of inserts as output. Thus, the filter can be made stateless also by using an index type of **`pi_EMPTY`**.

The compute window transforms the incoming inserts by selecting some of the output fields of the input events. The same window computes other fields based on values of the input event. It generates only inserts, so it can be stateless.

After the compute window, there is an aggregate window. This window type needs to retain events. Aggregate windows group data and compress groups into single events. If an aggregate window is fed a stream of Inserts, it would grow in an unbounded way.

To control this growth, you can connect the two sections of the model with a copy window with a retention policy.

Display 15.2 *Modified Event Stream Processing Model with Stateless and Stateful Parts*



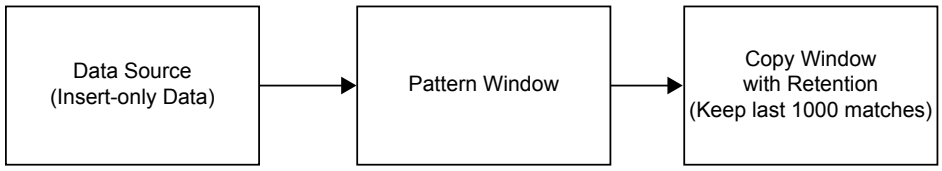
The stateful part of the model is accurately computed, based on the rolling window of input data. This model is bounded in memory.

Controlling Pattern Window Matches

Pattern matches that are generated by pattern windows are Inserts. Suppose you have a source window feeding a pattern window. Because a pattern window generate Inserts only, you should make it stateless by specifying an index type of **`pi_EMPTY`**. This prevents the pattern window from growing infinitely. Normally, you want to keep some of the more recent pattern matches around. Because you do not know how frequent the pattern generates matches, follow the pattern window with a count-based copy window.

Suppose you specify to retain the last 1000 pattern matches in the copy window.

Display 15.3 *Event Stream Processing Model with Copy with Retention*



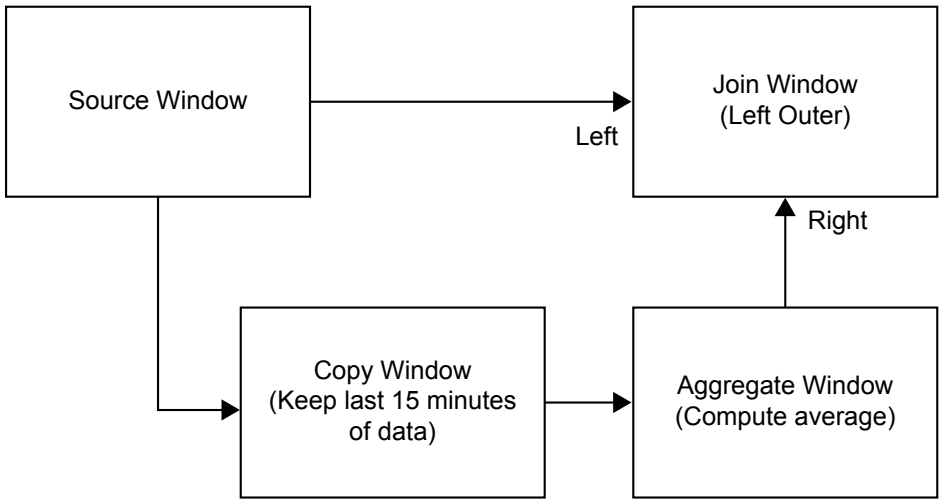
In cases like these, it is more likely that the copy window is queried from the outside using adapters, or publish/subcribe clients. The copy window might also feed other sections of the model.

Augmenting Incoming Events with Rolling Statistics

Suppose you have an insert stream of events, and one or more values are associated with the events. You want to augment each input event with some rolling statistics and then output the augmented events. Solving this problem requires using advanced features of the modeling environment.

For example, suppose you have a stream or stock trades coming in and you want to augment them with the average stock price in the past. You build the following model.

Display 15.4 *Event Stream Processing Model Using Advanced Features*



To control the aggregate window:

- Put retention before it (the copy window).
- Group it by symbol (which is bounded), and use the additive aggregation function average (**`ESP_aAve`**), which does not need to store each event for its computation.

The join window can be problematic. Ordinarily you think of a join window as stateful. A join retains data for its fact window or dimension window and performs matches. In this case, you want a special, but frequently occurring behavior. When input comes in, pause it at the join until the aggregate corresponding to the input is processed. Then link the two together, and pass the augmented insert out.

To process input in this fashion:

1. Make the join a left outer join, with the source feed the left window, and the aggregate feeds the right window.
2. Set Tagged Token data flow model on for the projects. This turns on a special feature that causes a fork of a single event to wait for both sides of the fork to rejoin before generating an output event.

3. Set the index type of the join to **`pi_EMPTY`**, making the join stateless. A stateless left outer join does not create a local copy of the left driving window (FACT window). It does not keep any stored results of the join. However, there is always a reference-counted copy of the lookup window. In the case of a left outer join, this is the right window. The lookup window is controlled by retention in this case, so it is bounded.
4. Ordinarily, a join, when the dimension window is changed, tries to find all matching events in the fact window and then issue updates for those joined matches. You do not want this behavior, because you are matching events in lock step. Further, it is simply not possible because you do not store the fact data. To prevent this regeneration on each dimension window change, set the no-regenerate option on the join window.

In this way you create a fast, lightweight join. This join stores only the lookup side, and produces a stream of inserts on each inserted fact event. A stateless join is possible for left and right outer joins.

Chapter 16

Advanced Topics

Logging Bad Events

When you start an event stream processing application with **`-b <filename>`**, the application writes the events that are not processed because of computational failures to

a log file. When you do not specify this option, the same data is output to **stderr**. It is recommended to create bad event logs so that they can be monitored for new insertions.

An application can use the **dfESPengine::logBadEvent()** member function from a procedural window to log events that it determines are invalid. For example, you can use the function to allow models to perform data quality checks and log events that do not pass. There are two common reasons to reject an event in a source window:

- The event contains a null value in one of the key fields.
- The opcode that is specified conflicts with the existing window index (for example, two Inserts of the same key, or a Delete of a non-existing key).

Measuring Time Granularity

Several methods in the C++ Modeling API 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.

Converting CSV Events to Binary

You can convert a file of CSV events into a file of binary events. This file can be read into a project and played back at higher rates than the CSV file.

CSV conversion is very CPU intensive. Convert events offline a single time to avoid the performance penalties of converting CSV events during each playback. In actual production applications, the data frequently arrives in some type of binary form and needs only reshuffling to be used in the SAS Event Stream Processing Engine. Otherwise, the data comes as text that needs to be converted to binary events.

For CSV conversion to binary, see the example application "csv2bin" under the **src** directory of the SAS Event Stream Processing Engine installation. The README file in **src** explains how to use this example to convert CSV files to event stream processor binary files. The source file shows you how to actually do the conversion in C++ using methods of the C++ Modeling API.

The following code example reads in binary events from **stdin** and injects the events into a running project. Note that only events for 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.

```
dfESPfileUtils::setBinaryMode(stdin);
// For windows it is essential that we 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.
```

Implementing Periodic (or Pulsed) Window Output

In most cases, the SAS Event Stream Processing EngineAPI is fully event driven. Windows continuously produce output as soon as they transform input. There are times when you might want a window to hold data and then output a canonical batch of updates. In this case, operations to common key values are collapsed into one operation.

Here are two cases where batched output might be useful:

- Visualization clients might want to get updates once a second, given that they cannot visualize changes any faster than this. When the event data is pulsed, the clients take advantage of the reduction of event data to visualize through the collapse around common key values.
- A window following the pulsed window is interested in comparing the deltas between periodic snapshots from that window.

Use the following call to add output pulsing to a window:

```
dfESPwindow::setPulseInterval(size_t us);
```

Note: Periodicity is specified in microseconds. However, given the clock resolution of most non-real-time operating systems, the minimum value that you should specify for a pulse period is 100 milliseconds.

Splitting Generated Events across Output Slots

Overview

All window types can register a splitter function or expression to determine what output slot or slots should be used for a newly generated event. This enables you to send generated events across a set of output slots.

Most windows send all generated events out of output slot 0 to zero of more downstream windows. For this reason, it is not standard for most models to use splitters. Using window splitters can be more efficient than using filter windows off a single output slot.

This is especially true, for example, when you are performing an alpha-split across a set of trades or a similar task.

Using window splitters is more efficient than using two or more subsequent filter windows. This is because the filtering is performed a single time at the window splitter rather than multiple times for each filter. This results in less data movement and processing.

Splitter Functions

Here is a prototype for a splitter function.

```
size_t splitterFunction(dfESPschema *outputSchema, dfESPeventPtr nev,
     dfESPeventPtr oev);
```

This splitter function receives the schema of the events supplied, the new and old event (only non-null for update block), and it returns a slot number.

Here is how you use the splitter for the source window (**sw_01**) to split events across three copy windows: **cw_01, cw_02, cw_03**.

```
sw_01->setSplitter(splitterFunction);
cq_01->addEdge(sw_01, 0,  cw_01);
cq_01->addEdge(sw_01, 1,  cw_02);
cq_01->addEdge(sw_01, -1, cw_03);
```

The **dfESPwindow::setSplitter()** member function is used to set the user-defined splitter function for the source window. The **dfESPcontquery::addEdge()** member function is used to connect the copy windows to different output slots of the source window.

When adding an edge between windows in a continuous query, specify the slot number of the parent window where the receiving window receives its input events. If the slot number is -1, it receives all the data produced by the parent window regardless of the splitter function.

If no splitter function is registered with the parent window, the slots specified are ignored, and each child window receives all events produced by the parent window.

Note: Do not write a splitter function that randomly distributes incoming records. Also, do not write a splitter function that relies on a field in the event that might change. The change might cause the updated event to generate a different slot value than what was produced prior to the update. This can cause an Insert to follow one path and a subsequent Update to follow a different path. This generates inconsistent results, and creates indices in the window that are not valid.

Splitter Expressions

When you define splitter expressions, you do not need to write the function to determine and return the desired slot number. Instead, the registered expression does this using the splitter expression engine. Applying expressions to the previous example would look as follows, assuming that you split on the field name "splitField", which is an integer:

```
sw_01->setSplitter("splitField%2");
cq_01->addEdge(sw_01, 0,  cw_01);
cq_01->addEdge(sw_01, 1,  cw_02);
cq_01->addEdge(sw_01, -1, cw_03);
```

Here, the **dfESPwindow::setSplitter()** member function is used to set the splitter expression for the source window. Using splitter expressions rather than

functions can lead to slower performance because of the overhead of expression parsing and handling. Most of the time you should not notice differences in performance.

dfESPwindow::setSplitter() has two additional optional parameters with defaults set to NULL.

- **initExp** enables you to specify an initialization expression for the expression engine used for this window's splitter.
- **initRetType** enables you to specify a return **datavar** value in those cases when you want to pass state from the initialization expression to the C++ application thread that makes the call. Most initialization expressions do not use return values from the initialization.

This initialization message enables you to specify some setup state, perhaps variable declarations and initialization, that you can use later in the splitter expression processing.

The full syntax for this call is as follows:

```
dfESPdatavarPtr setSplitter(const char* splitterExp, const char*
                            initExp=NULL, dfESPdatavar::dfESPdatatype
                            initRetType=dfESPdatavar::ESP_NULL);
```

You can find an example of window output splitter initialization in splitter_with_initexp in **$DFESP_HOME/src**. The example uses the following **setSplitter** call where the initialize declares and sets an expression engine variable to 1:

```
(void)sw_01->setSplitter("counter=counter+1; return counter%2",
                                        "integer counter\r\ncounter=1");
```

For each new event the initialize increments and mods the counter so that events rotate between slots 0 and 1.

Marking Events as Partial-Update on Publish

Overview

In most cases, events are published into an event stream processing engine with all fields available. Some of the field values might be null. Events with Delete opcodes require only the key fields to be non-null.

There are times when only the key fields and the fields being updated are desired or available for event updates. This is typical for financial feeds. For example, a broker might want to update the price or quantity of an outstanding order. You can update selected fields by marking the event as partial-update (rather than normal).

When you mark events as partial-update, you provide values only for the key fields and for fields that are being updated. In this case, the fields that are not updated are marked as data type **dfESPdatavar::ESP_LOOKUP**. This marking tells the SAS Event Stream Processing Engine to match key fields of an event retained in the system with the current event and not to update the current event's fields.

In order for a published event to be tagged as a partial-update, the event must contain all non-null key fields that match an existing event in the source window. Partial updates are applied to source windows only.

When using transactional event blocks that include partial events, be careful that all partial updates are for key fields that are already in the source window. You cannot include the insert of the key values with an update to the key values in a single event

block with transactional properties. This attempt fails and is logged because transactional event blocks are treated atomically. All operations in that block are checked against an existing window state before the transactional block is applied as a whole.

Publishing Partial Events into a Source Window

Consider these three points when you publish partial events into a source window.

- In order to construct the partial event, you must represent all the fields in the event. Specify either the field type and value or a placeholder field that indicates that the field value and type are missing. In this way, the existing field value for this key field combination remains for the updated event. These field values and types can be provided as **`datavars`** to build the event. Alternatively, they can be provided as a comma-separated value (CSV) string.

  If you use CSV strings, then use '^U' (such as, control-U, decimal value 21) to specify that the field is a placeholder field and should not be updated. On the other hand, if you use **`datavars`** to represent individual fields, then those fully specified fields should be valid. Enter them as **`datavars`** with values (non-null or null). Specify the placeholder fields as empty **`datavars`** of type **`dfESPdatavar::ESP_LOOKUP`**.

- No matter what form you use to represent the field values and types, the representation should be included in a call for the partial update to be published. In addition to the fields, use a flag to indicate whether the record is a normal or partial update. If you specify partial update, then the event must be an Update or an Upsert that is resolved to an Update. Using partial-update fields makes sense only in the context of updating an existing or retained source window event. This is why the opcode for the event must resolve to Update. If it does not resolve to Update, an event merge error is generated.

  If you use an event constructor to generate this binary event from a CSV string, then the beginning of that CSV string contains "**`u,p`**" to show that this is a partial-update. If instead, you use **`event->buildEvent()`** to create this partial update event, then you need to specify the event flag parameter as **`dfESPeventcodes::ef_PARTIALUPDATE`** and the event opcode parameter as **`dfESPeventcodes::eo_UPDATE`**.

- One or more events are pushed onto a vector and then that vector is used to create the event block. The event block is then published into a source window. For performance reasons, each event block usually contains more than a single event. When you create the event block you must specify the type of event block as transactional or atomic using **`dfESPeventblock::ebt_TRANS`** or as normal using **`dfESPeventblock::ebt_NORMAL`**.

  Do not use transactional blocks with partial updates. Such usage treats all events in the event block as atomic. If the original Insert for the event is in the same event block as a partial Update, then it fails. The events in the event block are resolved against the window index before the event block is applied atomically. Use normal event blocks when you perform partial Updates.

Examples

Here are some sample code fragments for the variations on the three points described in the previous section.

Create a partial Update **datavar** and push it onto the **datavar** vector.

```
// Create an empty partial-update datavar.
dfESPdatavar* dvp  = new dfESPdatavar(dfESPdatavar::ESP_LOOKUP);
// Push partial-update datavar onto the vector in the appropriate
// location.
// Other partial-update datavars might also be allocated and pushed to the
// vector of datavars as required.
dvVECT.push_back(dvp); // this would be done for each field in the update
     event
```

Create a partial Update using partial-update and normal **datavars** pushed onto that vector.

```
// Using the datavar vector partially defined above and schema,
// create event.
dfESPeventPtr eventPtr = new dfESPevent();
eventPtr->buildEvent(schemaPtr, dvVECT, dfESPeventcodes::eo_UPDATE,
dfESPeventcodes::ef_PARTIALUPDATE);
```

Define a partial update event using CSV fields where '**^U**' values represent partial-update fields. Here you are explicitly showing '**^U**'. However, in actual text, you might see the character representation of **Ctrl-U** because individual editors show control characters in different ways.

Here, the event is an Update (due to '**u**'), which is partial-update (due to '**p**'), key value is 44001, "ibm" is the instrument that did not change. The instrument is included in the field. The price is 100.23, which might have changed, and 3000 is the quantity which might have changed, so the last three of the fields are not updated.

```
p = new dfESPevent(schema_01,
(char *)"u,p,44001,ibm,100.23,3000,^U,^U,^U");
```

Understanding Primary Indexes and Retention Policies

Overview

Each window type has a primary index that is built upon creation of the window instance. The index is used to store events and enable rapid event lookup and retrieval. The **dfESPeventdepot_mem** object used to store windows supports six types of primary indices: five are stateful, and one is not.

Fully Stateful Indexes

The following index types are fully stateful:

Index Type	Description
pi_RBTREE	Specifies red-black tree, logarithmic Insert, Deletes are O(log(n)) — provides smooth latencies.

Index Type	Description
pi_HASH	Specifies a typical open hash algorithm. In general this index provides faster results than **pi_RBTREE**. Unless properly sized, using this index might lead to latency spikes.
pi_CL_HASH	Specifies a closed hash. This index provides faster results than **pi_HASH**.
pi_FW_HASH	Specifies a forward hash. This index creates a smaller memory footprint than other hash indexes, but might yield poorer delete performance.
pi_LN_HASH	Specifies a linked hash. This index performs slightly more slowly than other hash index and uses more memory than **pi_CL_HASH**.

For information about the closed hash, forward hash, and linked hash variants, see "Miscellaneous Container Templates" at **http://www.medownloads.com/download-Miscellaneous-Container-Templates-147179.htm**.

Events are absorbed, merged into the window's index, and a canonical version of the change to the index is passed to all output windows. Any window that uses a fully stateful index has a size equal to the cardinality of the unique set of keys, unless a time or size-based retention policy is enforced.

When no retention policy is specified, a window that uses one of the fully stateful indices acts like a database table or materialized view. At any point, it contains the canonical version of the event log. Because common events are reference-counted across windows in a project, you should be careful that all retained events do not exceed physical memory.

Use the Update and Delete opcodes for published events (as is the case with capital market orders that have a lifecycle such as create, modify, and close order). However, for events that are Insert-only, you must use window retention policies to keep the event set bound below the amount of available physical memory.

Retention Policies for Fully Stateful Indexes

Source or copy windows using a fully stateful index can have one of several types of retention policies specified. To specify a retention policy, use the **setRetentionParms()** method on the window. This method takes three parameters:

- the retention type
- the number of records or the time in seconds to use for retention
- a flag to specify the basis for the time (wall-clock-driven time versus time-field-driven time)

Retention type can be one of the following options:

Retention Type	Description
ret_NONE	This is the default. It is the same as if **setRetentionParms** was not called.

Retention Type	Description
ret_BYTIME_SLIDING	Specifies to purge continuously from the window when an event age reaches the limit as specified in the retention parameter.
ret_BYTIME_JUMPING	Specifies that when the amount of time since the last purge is equal to the time specified by the retention parameter, purge the entire set of window events. Then start the next time cycle.
ret_BYCOUNT_SLIDING	Specifies to purge continuously from the window when the event set size reaches the limit as specified by the retention parameter.
ret_BYCOUNT_JUMPING	Specifies that when the window event set size reaches the limit specified by the retention parameter, purge the entire window event set and start.

When events are deleted from a window because of a user-specified retention policy, the Delete or Update of that event generates a run-time error.

For more information, see the detailed class and method documentation available through **$DFESP_HOME/doc/html/index.html**.

Non-Stateful Index

The non-stateful index is a source window that can be set to use the index type **pi_EMPTY**. It acts as a pass-through for all incoming events.

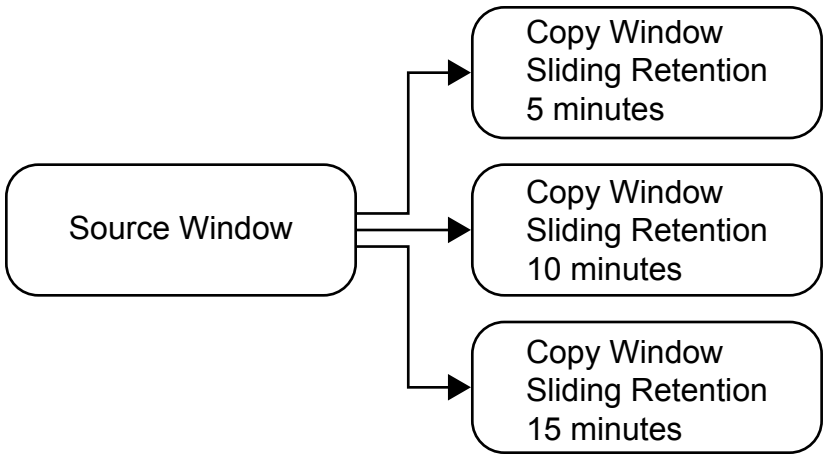
The following restrictions apply to source windows that use the empty index type.

- No restrictions apply if the source window is set to "Insert only.". For more information, see the **setInsertOnly** call in “dfESPwindow_source” on page 18.
- If the source window is not Insert-only, then it must be followed by a copy window with a stateful index. This restriction enables the copy window to resolve Updates, Upserts, and Deletes that require a previous window state. Otherwise, the Updates, Upserts, and Deletes are not properly handled and passed to subsequent derived windows farther down the model. As a result, the window cannot compute incremental changes correctly.

Using empty indices and retention enables you to specify multiple retention policies from common event streams coming in through a source window. The source window is

used as an absorption point and pass-through to the copy windows, as shown in the following figure.

Figure 16.1 *Copy Windows*



Using Aggregate Functions

Aggregate Functions for Aggregate Window Field Calculation Expressions

The following aggregate functions are available for aggregate window field calculation expressions:

Aggregate Function	Returns
`ESP_aAve(`*`fieldname`*`)`	average of the group
`ESP_aCount()`	number of events in the group
`ESP_aCountDistinct(`*`fieldname`*`)`	the number of distinct non-null values in the column specified by field name within a group
`ESP_aCountNonNull(`*`fieldname`*`)`	the number of events with non-null values in the column for the specified field name within the group
`ESP_aCountNull(`*`fieldname`*`)`	the number of events with null values in the column for the specified field name within the group
`ESP_aCountOpcodes(`*`opcode`*`)`	count of the number of events matching opcode for group
`ESP_aFirst(`*`fieldname`*`)`	the first event added to the group

Aggregate Function	Returns
ESP_aGUID()	a unique identifier
ESP_aLag(*fieldname,lag_value*)	a lag value where the following holds: • **ESP_aLag(*fieldname,*0) == ESPaLast(*fieldname*)** • **ESP_aLag(*fieldname,*1)** returns the second lag, which is the previous value of *fieldname* that affected the group
ESP_aLast(*fieldname*)	field from the last record that affected the group
ESP_aLastOpcodes(*opcode*)	the opcode of the last record to affect the group
ESP_aMax(*fieldname*)	maximum of the group
ESP_aMin(*fieldname*)	minimum of the group
ESP_aStd(*fieldname*)	standard deviation of the group
ESP_aSum(*fieldname*)	sum of the group
ESP_aWAve(*weight_fieldname, payload_fieldname*)	weighted group average

The following functions are always additive:

- **ESPaAve**
- **ESPaCount**
- **ESPaCountOpcodes**
- **ESPaCountDistinct**
- **ESPaGUID**
- **ESPaLastOpcode**
- **ESPaStd**
- **ESPaSum**
- **ESPaWAve**

The following functions are additive only when they get Inserts:

- **ESPaCountNonNull**
- **ESPaCountNull**
- **ESPaFirst**
- **ESPaLast**
- **ESPaMax**
- **ESPaMin**

You can implement a nonadditive **Max()** aggregate function using non-key field calculation expressions as follows:

```
dfESPwindow_aggregate *aw_01;
       aw_01 = cq->newWindow_aggregate("aggregateWindow_01", edm,
                                       dfESPindextypes::pi_RBTREE,
                                       aggr_schema);
       aw_01->addNonKeyFieldCalc("ESP_aSum(quantity)"); // sum(quantity)
       aw_01->addNonKeyFieldCalc("ESP_aMax(quantity)"); // max(quantity)
```

Using aggregate field expressions is simpler than aggregate functions, but they perform slower, and the number of functions is limited.

Note: In **ESP_aSum**, **ESP_aMax**, **ESP_aMin**, **ESP_aAve**, **ESP_aStd**, and **ESP_aWAve**, null values in a field are ignored. Therefore, they do not contribute to the computation.

The functions **ESP_aSum**, **ESP_aFirst**, **ESP_aWAve**, **ESP_aStd**, **ESP_aCount**, **ESP_aLast**, **ESP_aFirst**, **ESP_aLastNonDelete**, **ESP_aLastOpCode**, **ESP_aCountOpcodes** are all additive. That is, they can be calculated from retained state, and do not need to maintain a group state. This means that if these are the only functions used in a **dfESPwindow_aggrgate** instance, special optimizations are made and speed-ups of an order of magnitude in the aggregate window processing can occur.

The **dfESPgroupstate** class is used internally to maintain the groups in an aggregation and an instance of the **dfESPgroupstate** is passed to aggregate functions. The signature of an aggregate function is as follows:

```
typedef dfESPdatavarPtr (*dfESPaggregate_func)(dfESPschema *is,
                                    dfESPeventPtr nep, dfESPeventPtr oep,
                                    dfESPgroupstate *gs);
```

You can find this definition in the **api/dfESPfuncptr.h** file.

The **dfESPgroupstate** object does not only act as a container for a set of events belonging to the same group, but it also maintains a state vector of **dfESPdatavars**, one state vector per non-key field, that can be used by aggregate functions to store a field's state. This enables quick incremental aggregate function updates when a new group member arrives.

For more information, see "Using Aggregate Functions" on page 216.

Using an Aggregate Function to Add Statistics to an Incoming Event

You can use the **ESP_aLast(fieldName)** aggregate function to pass incoming fields into the aggregate event that is created. This can be useful to add statistics to events through the aggregate window without having to use an aggregate window followed by a join window. Alternatively, using a join window after an aggregate window joins the aggregate calculations or event to the same event that feeds into the aggregate window. But the results in that case might not be optimal.

For example, suppose that this is the incoming event schema:

```
"ID*:int64,symbol:string,time:datetime,price:double"
```

Suppose that with this incoming event schema, you want to add an aggregate statistic:

```
"ID*:int64,symbol:string,time:datetime,price:double,ave_price:double"
```

There, the average is calculated over the group with the same "symbol."

Alternatively, you can define a single aggregate stream, with the following schema:

```
"ID:int64,symbol*:string,time:datetime,price:double,ave_price:double"
```

Note: The group-by is the key of the aggregation, which is "symbol".

Next, use **dfESPwindow_aggregate::addNonKeyFieldCalc(*expression*)** to register the following aggregation functions for each non-key field of this window, which in this case are “ID,” “time,” “price,” and “ave_price”:

```
awPtr->addNonKeyFieldCalc("ESP_aLast(ID)”);
awPtr->addNonKeyFieldCalc("ESP_aLast(time)”);
awPtr->addNonKeyFieldCalc("ESP_aLast(price)”);
awPtr->addNonKeyFieldCalc("ESP_aAve(price)”);
```

Suppose that the following events come into the aggregate window:

```
insert: 1, "ibm", 09/13/2001T10:48:00, 100.00
insert: 2, "orc", 09/13/2001T10:48:01, 127.00
insert: 3, "ibm", 09/13/2001T10:48:02, 102.00
insert: 4, "orc", 09/13/2001T10:48:03, 125.00
insert: 5, "orc", 09/13/2001T10:48:04, 126.00
```

The aggregate stream produces the following:

```
insert: 1, "ibm", 09/13/2001T10:48:00, 100.00, 100.00
insert: 2, "orc", 09/13/2001T10:48:01, 127.00, 127.00
update: 3, "ibm", 09/13/2001T10:48:00, 102.00, 101.00
update: 4, "orc", 09/13/2001T10:48:01, 125.00, 126.00
update: 5, "orc", 09/13/2001T10:48:01, 126.00, 126.00
```

By using **aLast(*fieldname*)** and then adding the aggregate fields of interest, you can avoid the subsequent join window. This makes the modeling cleaner.

Using Additive Aggregation Functions

Aggregate functions that compute themselves based on previous field state and a new field value are called additive aggregation functions. These functions provide computational advantages over aggregate functions. For example, even though it is possible to use the **Sum()** aggregate function to iterate over the group and compute a new sum when a new group changes, faster results are obtained when you maintain the **Sum()** in a **dfESPdatavar** in the **dfESPgroupstate** object and increment or decrement the object by the incoming value, provided that the new event is an Insert, Update, or Delete. The function then adjusts this field state so that it is up-to-date and can be used again when another change to the group occurs.

Using a few helper functions that operate on numeric **dfESPdatavars** (found in the **dfESPdatavar_numerics.h**header file), the code to implement an additive **Sum()** aggregation function is:

```
dfESPdatavarPtr Sum(dfESPschema *is, dfESPeventPtr ep, dfESPeventPtr oep,
                            dfESPgroupstate *gs)
       {
        const int index = 2; // quantity
        dfESPptrVect<dfESPdatavarPtr> &state = gs->getStateVector();

        dfESPdatavar *rdv = NULL; // return value
        dfESPdatavar *iNdv = NULL; // input value (new record)
        dfESPdatavar *iOdv = NULL; // input value (old record)
        dfESPdatavar *sdv = NULL; //  current state (the old sum for the group)
```

```
// Get the new value out of the input record (insert or delete).

iNdv = new dfESPdatavar(is->getTypeIO(index)); // create new datavar
ep->copyByIntID(index, iNdv);        // load input value (new record)

// Get the old value out of the input record (update).

if (oep) {
    iOdv = new dfESPdatavar(is->getTypeIO(index)); // create new datavar
    oep->copyByIntID(index, iOdv);        // load input value(old record)
}

// If the state has never been set, set it to 0.0.

if (state.empty()) { // initialize state to 0.0
    dfESPdatavarPtr dv = new dfESPdatavar(is->getTypeIO(index));
    dv->setDouble(0.0);
    state.push_back(dv);
 }

 // Get the previous state.
 sdv = state[0]; // There is one state variable for summation, the old sum.

 // Create a new datavar to store the returned result.
 rdv = new dfESPdatavar(is->getTypeIO(index));

 // For a delete r = STATE - INCOMING VALUE.
 if (ep->getOpcode() == dfESPeventcodes::eo_DELETE) {
    dfESPdatavarNumerics::dv_subtract(rdv, sdv, iNdv);
 }

 // For an insert r = STATE + INCOMING VALUE.
 if (ep->getOpcode() == dfESPeventcodes::eo_INSERT) {
     dfESPdatavarNumerics::dv_add(rdv, sdv, iNdv);
 }

 // For an UPDATE block, if there is no OLD value, it is an insert, if
 // If there is an old value, then:
 // r = STATE - OLD INCOMING VALUE + NEW INCOMING VALUE
 if (ep->getOpcode() == dfESPeventcodes::eo_UPDATEBLOCK) {
     if (oep == NULL) { // treat just like an insert.
         dfESPdatavarNumerics::dv_add(rdv, sdv, iNdv);
     } else {
     dfESPdatavarNumerics::dv_subtract(rdv, sdv, iOdv);
     dfESPdatavarNumerics::dv_add(rdv, sdv, iNdv);
     }
  }

  dfESPdatavarNumerics::dv_assign(sdv, rdv); // sdv = rdv
  delete iNdv;
  if (iOdv)
      delete iOdv;
  return rdv;
 }
```

An additive aggregation function can be complex for two reasons:

- They must look at the current state (for example, the last computed state).
- They must evaluate the type of incoming event to make proper adjustments.

Some aggregate functions cannot be made additive, like **`Max()`**. Given a mix of Inserts, Updates, and Deletes, **`Max()`** must iterate over the group when an Update or Delete is received to correctly compute the aggregate value.

The code to implement the nonadditive **`Max()`** aggregate function is as follows:

```
// This is an example of a non-additive aggregation function.
// It loops over all events in the group to compute the maximum.

dfESPdatavarPtr maximumAggr(dfESPschema *is, dfESPeventPtr ep,
                            dfESPeventPtr oep, dfESPgroupstate *gs) {
    const int index = 3; // the price.
    dfESPdatavar *rdv, *dv; // dv will be our return value.
    rdv = new dfESPdatavar(dfESPdatavar::ESP_DOUBLE); // return value
    dv = new dfESPdatavar(dfESPdatavar::ESP_DOUBLE); // scratch value
    dfESPeventPtr gEv = gs->getFirst();

    while (gEv) {
       // Get the input argument out of the record.
       gEv->copyByIntID(index, dv); // get Event value into dv;
       // Is the return value NULL? (It has not been set.)
       if (rdv->isNull()) {
           if (!dv->isNull())
              rdv->setDouble(dv->getDouble());
           } else {
           // Check for a new max
           if (!dv->isNull()) {
              if (dv->getDouble() > rdv->getDouble()) // do we have a new max?
                  rdv->setDouble(dv->getDouble());
               }
            }
            gEv = gs->getNext();
         }
         delete dv;
         return rdv;
 }
```

After you specify aggregate functions, implementing **`dfESPwindow_aggregate`** is straightforward:

```
dfESPstring aggr_schema =
          dfESPstring
        ("symbol*:string,totalQuant:int32,maxPrice:double");
dfESPwindow_aggregate *aw_01;
aw_01 = cq->newWindow_aggregate("aggregateWindow01", edm,
     dfESPindextypes::pi_RBTREE, aggr_schema);
aw_01->addNonKeyFieldCalc(summationAggr, true);
// Add the summation function.
aw_01->addNonKeyFieldCalc(maximumAggr, false);
```

The final parameter of the **`addNonKeyFieldCalc`** method indicates whether the function being registered is additive. If all the aggregate functions unused in an aggregate window are additive, special optimizations are made. These optimizations yield significant performance advantages. In this optimized state, all of the events that

make up an aggregation group are not stored or indexed. The aggregate functions are all additive and do not need to look at an aggregation group.

Persist and Restore Operations

SAS Event Stream Processing Engine enables you to do the following:

- persist a complete model state to a file system
- restore a model from a persist directory that had been created by a previous persist operation
- persist and restore an entire engine
- persist and restore a project

To create a persist object for a model, provide a pathname to the class constructor: **dfESPpersist(char *_baseDir_);** The ***baseDir*** parameter can point to any valid directory, including disks shared among multiple running event stream processors.

After providing a pathname, call either of these two public methods:

```
bool persist();
bool restore(bool dumpOnly=false);
// dumpOnly = true means do not restore, just walk and print info
```

The **persist()** method can be called at any time. Be aware that it is expensive. Event block injection for all projects is suspended, all projects are quiesced, persist data is gathered and written to disk, and all projects are restored to normal running state.

The **restore()** method should be invoked only before any projects have been started. If the persist directory contains no persist data, the **restore()** call does nothing.

The persist operation is also supported by the C and Java publish/subscribe APIs. These API functions require a *host:port* parameter to indicate the target event stream processing engine.

The C publish/subscribe API method is as follows: **int C_dfESPpubsubPersistModel(char *hostportURL, const char *persistPath)**

The Java publish/subscribe API method is as follows: **boolean persistModel(String hostportURL, String persistPath)**

One application of the persist and restore feature is saving state across event stream processor system maintenance. In this case, the model includes a call to the **restore()** function described previously before starting any projects. To perform maintenance at a later time on the running engine:

1. Pause all publish clients in a coordinated fashion.
2. Make one client execute the publish/subscribe persist API call described previously.
3. Bring the system down, perform maintenance, and bring the system back up.
4. Restart the event stream processor model, which executes the **restore()** function and restores all windows to the states that were persisted in step 2.
5. Resume any publishing clients that were paused in step 1.

To persist an entire engine, use the following functions:

```
bool dfESPengine::persist(const char * path);
```

```
void dfESPengine::set_restorePath(const char *path);
```

The path that you specify for **persist** can be the same as the path that you specify for **set_restorePath**.

To persist a project, use the following functions:

```
bool dfESPproject::persist(const char *path)

bool dfESPproject::restore(const char *path);
```

Start an engine and publish data into it before you persist it. It can be active and receiving data when you persist it.

To persist an engine, **call dfESPengine::persist(*path*);**. The system does the following:

1. pauses all incoming messages (suspends publish/subscribe)
2. finish processing any queued data
3. after all queued data has been processed, persist the engine state to the specified directory, creating the directory if required
4. after the engine state is persisted, resume publish/subscribe and enable new data to flow into the engine

To restore an engine, initialize it and call **dfESPengine::set_restorePath(*path*);**. After the call to **dfESPengine::startProjects()** is made, the entire engine state is restored.

To persist a project call **dfESPproject::persist(*path*);**. The call turns off publish/subscribe, quiesces the system, persists the project, and then re-enables publish/subscribe. The path specified for restore is usually the same as that for persist.

To restore the project, call **dfESPproject::restore(*path*);** before the project is started. Then call **dfESPengine::startProject(*project*);**

Gathering and Saving Latency Measurements

The **dfESPlatencyController** class supports gathering and saving latency measurements on an event stream processing model. Latencies are calculated by storing 64-bit microsecond granularity timestamps inside events that flow through windows enabled for latency measurements.

In addition, latency statistics are calculated over fixed-size aggregations of latency measurements. These measurements include average, minimum, maximum, and standard deviation. The aggregation size is a configurable parameter. You can use an instance of the latency controller to measure latencies between any source window and some downstream window that an injected event flows through.

The latency controller enables you to specify an input file of event blocks. The rate at which those events are injected into the source window. It buffers the complete input file in memory before injecting to ensure that disk reads do not skew the requested inject rate.

Specify an output text file that contains the measurement data. Each line of this text file contains statistics that pertain to latencies gathered over a bucket of events. The number

of events in the bucket is the configured aggregation size. Lines contain statistics for the next bucket of events to flow through the model, and so on.

Each line of the output text file consists of three tab-separated columns. From left to right, these columns contain the following:

- the maximum latency in the bucket
- the minimum latency in the bucket
- the average latency in the bucket

You can configure the aggregation size to any value less than the total number of events. A workable value is something large enough to get meaningful averages, yet small enough to get several samples at different times during the run.

If publish/subscribe clients are involved, you can also modify publisher/subscriber code or use the file/socket adapter to include network latencies as well.

To measure latencies inside the model only:

1. Include "**int/dfESPlatencyController.h**" in your model, and add an instance of the **dfESPlatencyController** object to your **main()**.
2. Call the following methods on your **dfESPlatencyController** object to configure it:

Method	Description
void set_playbackRate(int32_t r)	Sets the requested inject rate.
void set_bucketSize(int32_t bs)	Sets the bucketSize parameter previously described.
void set_maxEvents(int32_t me)	Sets the maximum number of events to inject.
void set_oFile(char *ofile)	Sets the name of the output file containing latency statistics.
void set_iFile(char *ifile)	Sets the name of the input file containing binary event block data.

3. Add a subscriber callback to the window where you would like the events to be timestamped with an ending timestamp. Inside the callback add a call to this method on your **dfESPlatencyController** object: **void record_output_events(dfESPeventblock *ob)**. This adds the ending timestamp to all events in the event block.
4. After starting projects call these methods on your **dfESPlatencyController** object:

Method	Description
`void set_injectPoint(dfESPwindow_source *s)`	Sets the source window in which you want events time stamped with a beginning timestamp.
`void read_and_buffer()`	Reads the input event blocks from the configured input file and buffers them.
`void playback_at_rate()`	Time stamps input events and injects them into the model at the configured rate, up to the configured number of events.

5. Quiesce the model and call this method on your **`dfESPlatencyController`**object: **`void generate_stats()`**. This writes the latency statistics to the configured output file.

To measure model and network latencies by modifying your publish/subscribe clients:

1. In the model, call the **`dfESPengine setLatencyMode()`** function before starting any projects.
2. In your publisher client application, immediately before calling **`C_dfESPpublisherInject()`**, call **`C_dfESPlibrary_getMicroTS()`** to get a current timestamp. Loop through all events in the event block and for each one call **`C_dfESPevent_setMeta(event, 0, timestamp)`** to write the timestamp to the event. This records the publish/subscribe inject timestamp to meta location 0.
3. The model inject and subscriber callback timestamps are recorded to meta locations 2 and 3 in all events automatically because latency mode is enabled in the engine.
4. Add code to the inject loop to implement a fixed inject rate. See the latency publish/subscribe client example for sample rate limiting code.
5. In your subscriber client application, include "**`int/dfESPlatencyController.h`**" and add an instance of the **`dfESPlatencyController`** object.
6. Configure the latency controller bucketSize and playbackRate parameters as described previously.
7. Pass your latency controller object as the context to **`C_dfESPsubscriberStart()`** so that your subscriber callback has access to the latency controller.
8. Make the subscriber callback pass the latency controller to **`C_dfESPlatencyController_recordExtraOutputEvents()`**, along with the event block. This records the publish/subscribe callback timestamp to meta location 4.
9. When the subscriber client application has received all events, you can generate statistics for latencies between any pair of the four timestamps recorded in each event. First call **`C_dfESPlatencyController_setOFile()`** to set the output file. Then write the statistics to the file by calling **`C_dfESPlatencyController_generateStats()`** and passing the latency

controller and the two timestamps of interest. The list of possible timestamp pairs and their time spans are as follows:

- (0, 2) – from inject by the publisher client to inject by the model
- (0, 3) – from inject by the publisher client to subscriber callback by the model
- (0, 4) – from inject by the publisher client to callback by the subscriber client (full path)
- (2, 3) – from inject by the model to subscriber callback by the model
- (2, 4) – from inject by the model to callback by the subscriber client
- (3, 4) – from subscriber callback by the model to callback by the subscriber client

10. To generate further statistics for other pairs of timestamps, reset the output file and call **`C_dfESPlatencyController_generateStats()`** again.

To measure model and network latencies by using the file/socket adapter, run the publisher and subscriber adapters as normal but with these additional switches:

Publisher	
—r *rate*	Specifies the requested transmit rate in events per second.
-m *maxevents*	Specifies the maximum number of events to publish.
-p	Specifies to buffer all events prior to publishing.
-n	Enables latency mode.

Subscriber	
-r *rate*	Specifies the requested transmit rate in events per second.
-a *aggrsize*	Specifies the aggregation bucket size.
-n	Enables latency mode.

The subscriber adapter gathers all four timestamps described earlier for the windows specified in the respective publisher and subscriber adapter URLs. At the end of the run, it writes the statistics data to files in the current directory. These files are named "latency_**`transmit rate_high timestamp_low timestamp`**", where the high and low timestamps correspond to the timestamp pairs listed earlier.

Publish/Subscribe API Support for Google Protocol Buffers

Overview to Publish/Subscribe API Support for Google Protocol Buffers

SAS Event Stream Processing Engine provides a library to support Google Protocol Buffers. This library provides conversion methods between an SAS Event Stream Processing Engine event block in binary format and a serialized Google protocol buffer (**protobuf**).

To exchange a **protobuf** with an event stream processing server, a publish/subscribe client using the standard publish/subscribe API can call **C_dfESPpubsubInitProtobuff()** to load the library that supports Google Protocol Buffers. Then a publisher client with source data in **protobuf** format can call **C_dfESPprotobuffToEb()** to create event blocks in binary format before calling **C_dfESPpublisherInject()**. Similarly, a subscriber client can convert a received events block to a **protobuf** by calling **C_dfESPebToProtobuff()** before passing it to a **protobuf** handler.

Note: The server side of an event stream processing publish/subscribe connection does not support Google Protocol Buffers. It continues to send and receive event blocks in binary format only.

The SAS Event Stream Processing Engine Java publish/subscribe API contains a **protobuf** JAR file that implements equivalent methods for Java publish/subscribe clients. In order to load the library that supports Google Protocol Buffers, you must have installed the standard Google Protocol Buffers run-time library. The SAS Event Stream Processing Engine run-time environment must be able to find this library. For Java, you must have installed the Google **protobuf** JAR file and have included it in the run-time class path.

A publish/subscribe client connection exchanges events with a single window using that window's schema. Correspondingly, a **protobuf** enabled client connection uses a single fixed **protobuf** message type, as defined in a message block in a **.proto** file. The library that supports Google Protocol buffers dynamically parses the message definition, so no precompiled message-specific classes are required. However, the Java library uses a **.desc** file instead of a **.proto** file, which requires you to run the Google **protoc** compiler on the **.proto** file in order to generate a corresponding **.desc** file.

For C clients, the name of the **.proto** file and the enclosed message are both passed to the library that supports Google Protocol Buffers in the **C_dfESPpubsubInitProtobuff()** call. This call returns a **protobuf** object instance, which is then passed in all subsequent **protobuf** calls by the client. This instance is specific to the **protobuf** message definition. Thus, it is valid as long as the client connection to a specific window is up. When the client stops and restarts, it must obtain a new **protobuf** object instance.

For Java clients, the process is slightly different. The client creates an instance of a **dfESPprotobuf** object and then calls its **init()** method. Subsequent **protobuf** calls are made using this object's methods, subject to the same validity scope described for the C++ **protobuf** object.

Conversion between a binary event block and a **protobuf** is accomplished by matching fields in the **protobuf** message definition to fields in the schema of the associated

publish/subscribe window. Ensure that the **protobuf** message definition and the window schema are compatible.

The following mapping of event stream processor to Google Protocol Buffer data types are supported:

Event Stream Processor Data Type	Google Protocol Buffer Data Type
ESP_DOUBLE	DOUBLE FLOAT TYPE_DOUBLE TYPE_FLOAT
ESP_INT64	INT64 FIXED64 SFIXED64 SINT64 TYPE_INT64 UINT64
ESP_INT32	ENUM INT32 FIXED32 SFIXED32 SINT32 TYPE_INT32 UINT32
ESP_UTF8STR	STRING TYPE_STRING
ESP_DATETIME	INT64 TYPE_INT64
ESP_TIMESTAMP	INT64 TYPE_INT64
Other mappings are currently unsupported.	

Converting Nested and Repeated Fields in Protocol Buffer Messages to an Event Block

Provided that they are supported, you can repeat the message fields of a **protobuf**. A message field of TYPE_MESSAGE can be nested, and possibly repeated as well. All of these cases are supported when converting a **protobuf** message to an event block, observing the following policies:

- A **protobuf** message that contains nested messages requires that the corresponding schema be a flattened representation of the **protobuf** message. For example, a **protobuf** message that contains three fields where the first is a nested message with four fields, the second is not nested, and the third is a nested message with two fields requires a schema with 4 + 1 + 2 = 7 fields. Nesting depth is unbounded.
- A single **protobuf** message is always converted to an event block that contains a single event, provided that no nested message field is repeated.
- When a **protobuf** message has a non-message type field that is repeated, all the elements in that field are gathered into a single comma-separated string field in the event. For this reason, any schema field that corresponds to a repeated field in a **protobuf** message must have type ESP_UTF8STR, regardless of the field type in the **protobuf** message.
- A **protobuf** message that contains nested message fields that are repeated is always converted to an event block that contains multiple events. There is one event for each element in every nested message field that is repeated.

Converting Event Blocks to Protocol Buffer Messages

Converting an event block to a **protobuf** is conceptually similar to converting nested and repeated fields in a **protobuf** to an event block, but the process requires more code. Every event in an event block is converted to a separate **protobuf** message. For this reason, the **C_dfESPebToProtobuff()** library call takes an index parameter that indicates which event in the event block to convert. The library must be called in a loop for every event in the event block.

The conversion correctly loads any nested fields in the resulting **protobuf** message. Any repeated fields in the resulting **protobuf** message contain exactly one element, because event blocks do not support repeated fields.

Note: Event block to **protobuf** conversions support only events with the Insert opcode, because event opcodes are not copied to **protobuf** messages. Conversions of **protobuf**to event blocks always use the Upsert opcode.

Support for Transporting Google Protocol Buffers through a Solace Appliance

Support for Google Protocol Buffers is available through the Solace connector and adapter. The adapter supports transport of a **protobuf** through the appliance, instead of binary event blocks. This allows a third-party publisher or subscriber to connect to a Solace appliance and exchange a **protobuf** with an engine without using the publish/subscribe API. The **protobuf** message format and window schema must be compatible.

The Solace connector and adapter requires configuration of the **.proto** file and message name through the **protofile** and **protomsg** parameters. The connector converts a **protobuf** to and from an event block using the SAS Event Stream Processing Engine library that supports Google Protocol Buffers. In addition, the C and Java Solace publish/subscribe clients also support Google Protocol Buffers when configured to do so in the solace.cfg client configuration file. A **protobuf** enabled publisher converts an event block to a **protobuf** to transport through the appliance to a third-party consumer of Google Protocol Buffers. Likewise, a **protobuf**-enabled subscriber receives a **protobuf** from the appliance and converts it to an event block.

Appendix 1

Example: Using Blue Fusion Functions

The following example creates a compute window that uses the Blue Fusion standardize function. The function normalizes the **City** field that is created for events in that window.

This example provides a general demonstration of how to use Blue Fusion functions in SAS Event Stream Processing Engine expressions. To use these functions, you must have installed the SAS DataFlux QKB (Quality Knowledge Base) product and set two environment variables: **DFESP_QKB** and **DFESP_QKB_LIC**. For more information, see "Using Blue Fusion Functions" on page 10.

```
// -*- Mode: C++; indent-tabs-mode: nil; c-basic-offset: 4 -*-

#include "dfESPengine.h"  // this also includes deESPlogUtils.h
#include "dfESPstring.h"
#include "dfESPevent.h"
#include "dfESPwindow_source.h"
#include "dfESPwindow_compute.h"
#include "dfESPeventdepot_mem.h"
#include "dfESPcontquery.h"
#include "dfESPeventblock.h"
#include "dfESPproject.h"

#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <cstdio>
#include <iostream>
#define SYMBOL_INDX 1
#define PRICE_INDX  2

// The compute window contains a field expression using a data quality function
// in the Blue Fusion library. It standarizes the city name.
//
// In order to run this example you need to download the DataFlux Quality
// Knowledge Base and set the environment variable DFESP_QKB to the root node
// of that install.

using namespace std;
void winSubscribe_compute(dfESPschema *os, dfESPeventblockPtr ob, void *ctx) {
     dfESPengine::oStream() << endl;
 << "--------------------------------------------------------------------------" <<
     endl;
dfESPengine::oStream() << "computeWindow" << endl;
ob->dump(os);
```

```
}
int main(int argc, char *argv[]) {

   // Call Initialize without overriding the framework defaults.
   dfESPengine *myEngine = dfESPengine::initialize(argc, argv, "myEngine",
                                                   pubsub_DISABLE);
     if (!myEngine) {
       cerr <<"Error: dfESPengine::initialize failed using all framework defaults\n";
     return 1;     }

   dfESPproject  *project;
   project = myEngine->newProject("project");

   dfESPcontquery  *contQuery;
   contQuery = project->newContquery("contquery");

   // Build the memory depot, source window schema, source window, copy window,
   //    and continous query objects.
   dfESPeventdepot_mem* depot;
   depot = project->newEventdepot_mem("memDepot");

   dfESPwindow_source *sw;
   sw = contQuery->newWindow_source("sourceWindow", depot, dfESPindextypes::pi_HASH,
                                    dfESPstring("name:string,ID*:int32,city:string"));
   dfESPschema  *sw_schema = sw->getSchema();

   dfESPwindow_compute *cw;
   cw = contQuery->newWindow_compute("computeWindow", depot, dfESPindextypes::pi_HASH,
           dfESPstring("ID*:int32,name:string,oldCity:string,newCity:string"));

   // Register the non-key field calculation expressions.
   // They must be added in the same non-key field order as the schema.
   cw->addNonKeyFieldCalc("name");  // pass name through unchanged
   cw->addNonKeyFieldCalc("city");  // pass city through unchanged

   // Run city through the blue fusion standardize function.
    char newCity[1024] = "bluefusion bf\r\n";
    strcat(newCity, "String result\r\n");
    strcat(newCity, "bf = bluefusion_initialize()\r\n");
    strcat(newCity, "if (isnull(bf)) then\r\n");
    strcat(newCity, "    print(bf.getlasterror())\r\n");
    strcat(newCity, "if (bf.loadqkb(\"ENUSA\") == 0) then\r\n");
    strcat(newCity, "    print(bf.getlasterror())\r\n");
    strcat(newCity, "if (bf.standardize(\"City\",city,result) == 0) then\r\n");
    strcat(newCity, "    print(bf.getlasterror())\r\n");
    strcat(newCity, "return result");
    cw->addNonKeyFieldCalc(newCity);

    // Add the subscriber callbacks to all the windows
    cw->addSubscriberCallback(winSubscribe_compute);

    // Add window connectivity
    contQuery->addEdge(sw, 0, cw);

    // create and start the project
    project->setNumThreads(2);
```

```
    myEngine->startProjects();

    // declare some variables to build up the input data.
    dfESPptrVect<dfESPeventPtr> trans;
    dfESPevent *p;

    // Insert multiple events
    p = new dfESPevent(sw_schema,(char *)"i,n,Jerry, 1111, apex");
    trans.push_back(p);
    p = new dfESPevent(sw_schema,(char *)"i,n,Scott, 1112, caryy");
    trans.push_back(p);
    p = new dfESPevent(sw_schema,(char *)"i,n,someone, 1113, rallleigh");
    trans.push_back(p);
    dfESPeventblockPtr ib = dfESPeventblock::newEventBlock(&trans,
                                            dfESPeventblock::ebt_TRANS);
    project->injectData(contQuery, sw, ib);

    // Inject the event block into the graph
    trans.free();
    project->quiesce();
    dfESPengine::shutdown();
    return 0;
}
```

Appendix 2

Setting Logging Level for Adapters

Logging levels for adapters use the same range of levels that you can set for the `C_dfESPpubsubInit()` publish/subscribe API call and in the engine `initialize()` call.

Table A2.1 *Logging Level for the Adapter*

Logging Level	Parameter Setting	Description	Where Appears
TRACE	dfESPLLTrace	Provide the most detailed information.	Logs
DEBUG	dfESPLLDebug	Provide detailed information about the flow through the system.	Logs
INFO	dfESPLLInfo	Report on run-time events of interest.	Immediately visible on the console
WARN	dfESPLLWarn	Indicates use of deprecated APIs, poor use of an API, or other run-time situations that are undesirable.	Immediately visible on the console
ERROR	dfESPLLError	Report on other run-time errors or unexpected conditions.	Immediately visible on the console
FATAL	dfESPLLFatal	Report severe errors that cause premature termination.	Immediately visible on the console
OFF	dfESPLLOff	Logging is turned off.	NA

Glossary

derived windows
windows that display events that have been fed through other windows and that perform computations or transformations on these incoming events.

directed graph
a set of nodes connected by edges, where the edges have a direction associated with them.

engine
the top-level container in a model. The engine manages the project resources.

event block
a grouping or package of events.

event stream processing
a process that enables real-time decision making by continuously analyzing large volumes of data as it is received.

event streams
a continuous flow of events (or more precisely, event blocks).

factory server
a server for factory objects that control the creation of other objects, access to other objects, or both. A factory object has a method for every object that it can create.

memory depot
a repository for indexes and event data that is used by a project.

modeling API
an application programming interface that enables developers to write event stream processing models.

operation code (opcode)
an instruction that specifies an action to be performed.

publish/subscribe API
a library that enables you to publish event streams into an event stream processor, or to subscribe to event streams, within the event stream processing model. The publish/subscribe API also includes a C and JAVA event stream processing object support library.

source window
a window that has no windows feeding into it and is the entry point for publishing events into the continuous query.

stream
a sequence of data elements made available over time.

thread pool
a set of threads that can be used to execute tasks, post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.

window
a processing node in an event stream processing model. Windows in a continuous query can perform aggregations, computations, pattern-matching, and other operations.