



THE
POWER
TO KNOW.

SAS[®] Event Stream Processing Engine 2.1

User's Guide

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2013. *SAS® Event Stream Processing Engine 2.1: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Event Stream Processing Engine 2.1: User's Guide

Copyright © 2013, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

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

September 2013

SAS provides a complete selection of books and electronic products to help customers use SAS® software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit **support.sas.com/bookstore** or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

<i>Using This Book</i>	<i>vii</i>
<i>What's New in SAS Event Stream Processing Engine</i>	<i>ix</i>
<i>Recommended Reading</i>	<i>xv</i>
Chapter 1 • Overview to SAS Event Stream Processing Engine	1
Product Overview	1
Conceptual Overview	2
Implementing Engine Models	4
Understanding Continuous Queries	5
Understanding Events	7
Understanding Event Blocks	8
Getting Started with SAS Event Stream Processing Engine	9
Writing an Application with SAS Event Stream Processing Engine ..	10
Chapter 2 • Programming with the C++ Modeling API	13
Using Expressions	14
Overview to the C++ Modeling API	16
Dictionary	17
Chapter 3 • Using the XML Modeling Layer	51
XML Modeling Layer	52
Using the XML Factory Server	56
Examples	57
Chapter 4 • Creating Pattern Windows	67
Overview of Pattern Windows	67
State Definitions for Operator Trees	70
Restrictions on Patterns	73
Example: Simple Pattern Window	75
Chapter 5 • Creating Aggregate Windows	81
Overview to Aggregate Windows	81

Flow of Operations	82
Aggregate Functions	83
Chapter 6 • Creating Procedural Windows	85
Overview to Procedural Windows	85
C++ Window Handlers	86
DS2 Window Handlers	89
Chapter 7 • Visualizing Event Streams	95
Overview to Event Visualization	95
Using Streamviewer	95
Using SAS/GRAPH	97
Chapter 8 • Using the Publish/Subscribe API	99
Overview to the API	99
Understanding Publish/Subscribe API Versioning	100
Using Callback Functions	101
The API from the Engine's Perspective	102
The API from the Client's Perspective	103
Functions for the Publish/Subscribe API	105
Using the Java Publish/Subscribe API	117
Chapter 9 • Using Connectors and Adapters	123
Using Connectors	124
Using Adapters	155
Chapter 10 • Enabling Guaranteed Delivery	175
Overview to Guaranteed Delivery	175
Guaranteed Delivery Success Scenario	178
Guaranteed Delivery Failure Scenarios	179
Additions to the Publish/Subscribe API for Guaranteed Delivery . .	180
Configuration File Contents	180
Publish/Subscribe API Implementation of Guaranteed Delivery . . .	181
Chapter 11 • Implementing 1+N-Way Failover	185
Overview to 1+N-Way Failover	186

Topic Naming	190
Failover Mechanisms	191
Restoring Failed Active ESP State after Restart	195
Using ESP Persist/Restore	196
Metadata Exchanges (Solace)	197
Metadata Exchanges (Tervela)	197
Required Software Components	198
Required Client Configuration	198
Required Appliance Configuration (Solace)	199
Required Appliance Configuration (Tervela)	200
Chapter 12 • Advanced Topics	201
Logging Bad Events	202
Measuring Time Granularity	202
Using Joins and Understanding Join Constraints	203
Converting CSV Events to Binary	204
Implementing Periodic (or Pulsed) Window Output	205
Splitting Generated Events across Output Slots	206
Marking Events as Partial-Update on Publish	208
Understanding Primary Indexes and Retention Policies	211
Using Stateless Pattern Windows	214
Using Aggregation Functions	215
Using an Aggregate Function to Add Statistics to an Incoming Event	218
Persist and Restore Model State	220
Gathering and Saving Latency Measurements	221
Appendix 1 • Example: Implementation of the Trades Model	227
Appendix 2 • Example: Subscriber and Publisher Applications	231
Appendix 3 • Example: Using Blue Fusion Functions	241
Appendix 4 • Setting Logging Level for Adapters	245
Glossary	247

Using This Book

Audience

This document provides information for programmers to use the 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) documentation that is shipped with the product.

What's New

What's New in SAS Event Stream Processing Engine

Overview

SAS Event Stream Processing Engine 2.1 provides the following new functionality:

- integration between DataFlux and SAS
- procedural window access to SAS analytics through DS2 scripting
- new adapters and connectors
- user-defined plug-in connectors
- new aggregate functions
- pre-allocated hash indexes
- 1+N-Way Failover
- Streamviewer metadata search
- engine state persist and recovery
- XML factory server project control
- Microsoft Windows 64-bit support for XML modeling

x *What's New in SAS Event Stream Processing Engine*

- latency performance measurements support

SAS and DataFlux Integration

SAS has fully integrated its DataFlux suite of data quality, data integration, data governance, and master data management solutions. These products help you build a more integrated information management approach that goes beyond data management and governance to support analytics and decision management.

Rather than being licensed separately, SAS DataFlux products are now being combined into software offerings that include other SAS products as well. Changes in offerings might affect your license renewals. Some DataFlux products are being rebranded and enhanced as SAS products, such as SAS Event Stream Processing Engine.

The [SAS Customer Support](#) site now provides support and resources for both SAS and SAS DataFlux products. The MyDataFlux portal is no longer supported. Some of these revised resources are:

- [SAS Downloads](#) site to download SAS DataFlux products, data updates, and Quality Knowledge Bases
- [SAS Documentation](#) site and [SAS Install Center](#) to find current documentation.
- [SAS Administrative Services](#) to manage your account, including how to contact Contracts Support to obtain or renew a license
- [SAS Support](#) to submit questions or a request for technical support.

Procedural Window Access to SAS Analytics through DS2 Scripting

You now can write a procedural window event stream handler in the DS2 language. Declare the program as a character string and set it in the procedural windows context.

New input handlers can take advantage of the SAS Threaded Kernel library that is embedded within the SAS Event Stream Processing Engine.

Using both the SAS Threaded Kernel library and DS2, you can essentially run existing SAS models using the engine to process events. SAS models with additive functionality, such as transaction scoring for fraud, can take advantage of this feature. It is important that input handlers for event streams process one event at a time without having to re-process events.

For more information, see [“Overview of DS2 Window Handlers” on page 89](#). For more information about DS2, see *SAS DS2 Language Reference*.

New Adapters and Connectors

New publish and subscribe adapters and connectors are provided to support the following products:

- IBM WebSphere MQ
- Tervela Data Fabric
- Solace Systems High-Performance Messaging Appliances
- Tibco Rendezvous (Tibco RV)
- OSIssoft PI (for publish only)
- Syslogs (as an extension to the file and socket adapter and connector)

For more information, see [Chapter 9, “Using Connectors and Adapters,” on page 123](#).

User-Defined Plug-in Connectors

Users can now write their own connectors and plug them into the SAS Event Stream Processing Engine platform. This capability had been available for adapters, which are stand-alone executables that use the publish and subscribe API. Connectors are C++

classes that required additional plug-in support from the SAS Event Stream Processing Engine platform for this capability to be provided.

For more information, see [“User-Written Connectors” on page 152](#).

New Aggregate Functions

Two new aggregate functions are available in SAS Event Stream Processing Engine 2.1:

- `ESP_aFirst()` returns the first event field value published for a given group, regardless of retention policy
- `ESP_aGUID()` generates globally unique identifiers

For more information, see [“Aggregate Functions” on page 83](#).

Pre-allocated Hash Indexes

SAS Event Stream Processing Engine now provides the ability for windows and depots to pre-allocate hash index size. This can be useful for latency performance improvements, where growth in hash indices during runtime visibly impacts latency measurements. Derived join indices and caches are also adjusted based on settings on their input windows. For more information, see [“Fully Stateful Indexes” on page 211](#).

1+N-way Failover

The SAS Event Stream Processing Engine now supports 1+N-Way Failover. This means that in addition to the primary event stream processing (ESP) server, you can configure one or more active failover ESP instances. This requires the use of a third-party messaging fabric. Currently, both Solace Systems and Tervela are supported.

This feature provides seamless failover for publishing and subscribing clients when primary event stream processing fails over to the next available recovery system. For more information, see [“Implementing 1+N-Way Failover” on page 186](#).

Streamviewer Metadata Search

The Streamviewer client provided with the event stream package for subscribing to and viewing event streams has been extended to enable metadata searches to determine what streams to view. Before, you needed to know the entire window URL in order to subscribe to an event stream processing window. Now you can query various objects available within an engine, starting with the projects and working down to windows to determine the subscription of interest.

For more information, see [“Using Streamviewer” on page 95](#).

Engine State Persist and Recovery

The SAS Event Stream Processing Engine now has the ability to persist to and recover from event stream processing engine state files. These capabilities are made available through the publish and subscribe APIs as well as through the C++ Modeling API.

The persist capability halts all input source queues. It then temporarily disables the continuous query graphs and writes engine state out in a hierarchy of files down to the window level. An engine must be restored in its entirety.

This feature can be useful for storing current state for an ESP engine before bringing it down for maintenance. The engine’s previous state can be restored after it is brought back on line. This feature is also useful for bringing a previously failed primary engine back on line as a newly available failover engine. For more information, see [“Persist and Restore Model State” on page 220](#).

XML Factory Server Project Control

The XML factory server, `dfESP_xml_server`, now supports the ability to start, stop, create, remove, inject, and query projects within a running event stream processing engine. It also now supports limited queries, event injects, and short-lived projects. For more information, see [“Using the XML Factory Server” on page 56](#).

Microsoft Windows 64-Bit Support for XML Modeling

The XML modeling factory server has now been made available on Microsoft Windows 64-bit platforms.

Latency Performance Measurements

Event stream processing latency measurements using the C++ Modeling API class are now supported through `dfESPLatencyController`. Additional switches are available to the file and socket adapter. This enables the injection of up to four timestamps with microsecond granularity to measure ESP processing latencies.

Recommended Reading

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

- *SAS Event Stream Processing Engine 2.1: Overview* provides an introduction to the product and an illustrative example.
- *SAS Event Stream Processing Engine 2.1: User's Guide* describes the product and provides technical details for writing event stream processing applications.
- Navigate to `$DFESP_HOME/doc/html` for detailed class and method documentation for the C++ modeling, C, and Java™ client publish/subscribe APIs. Specifically, documentation about the following topics is provided:
 - C++ Modeling API
 - Connector API
 - C Publish/Subscribe API
 - C wrapper over Modeling API (to support publishers and subscribers to handle event streams)
 - Java Publish and Subscribe API and wrapper over Modeling API (to support publishers and subscribers to handle event streams)

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

Overview to SAS Event Stream Processing Engine

<i>Product Overview</i>	1
<i>Conceptual Overview</i>	2
<i>Implementing Engine Models</i>	4
<i>Understanding Continuous Queries</i>	5
<i>Understanding Events</i>	7
<i>Understanding Event Blocks</i>	8
<i>Getting Started with SAS Event Stream Processing Engine</i>	9
Installing and Configuring SAS Event Stream Processing Engine	9
Using the SAS Event Stream Processing Engine	9
<i>Writing an Application with SAS Event Stream Processing Engine</i>	10

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 an XML factory 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

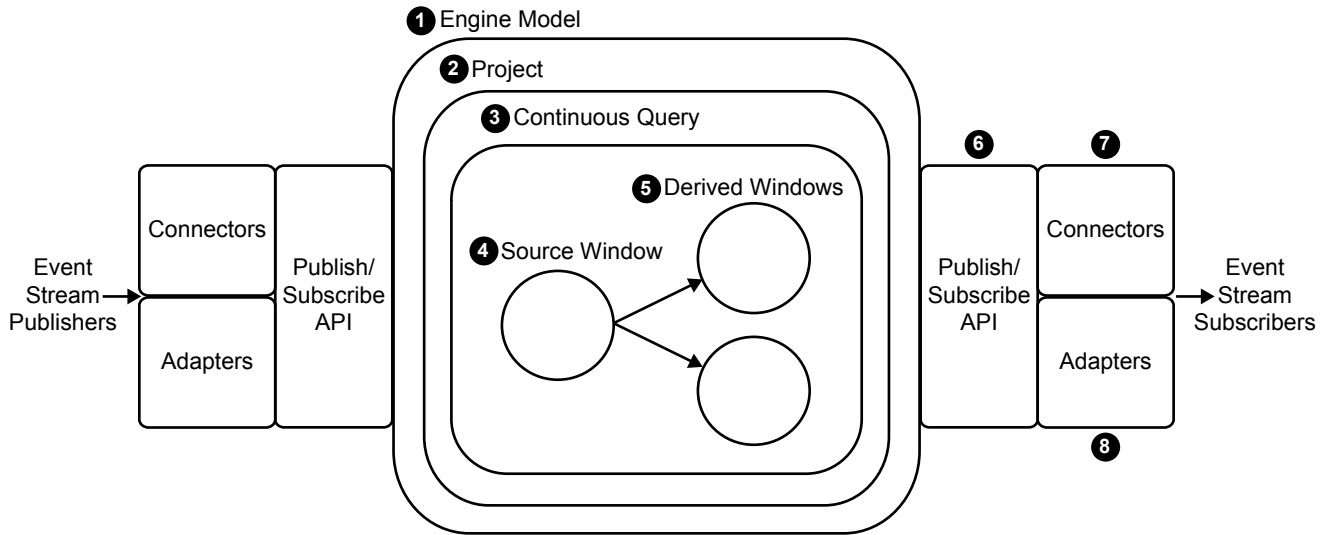
SAS Event Stream Processing Engine enables a programmer to write event stream processing applications that continuously analyze events in motion. When designing an application, programmers must answer the following questions:

- 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*.

- 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

Event stream processing engines can be XML factory servers or can be embedded within application processes using the C++ Modeling API. The application process that contains the engine can be a server shell, or it can be a working application thread that interacts with the engine threads. 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

Whether you implement multiple projects or multiple continuous queries depends on your processing needs. Create multiple projects when you want more than one threading model in an engine. 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 are 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.

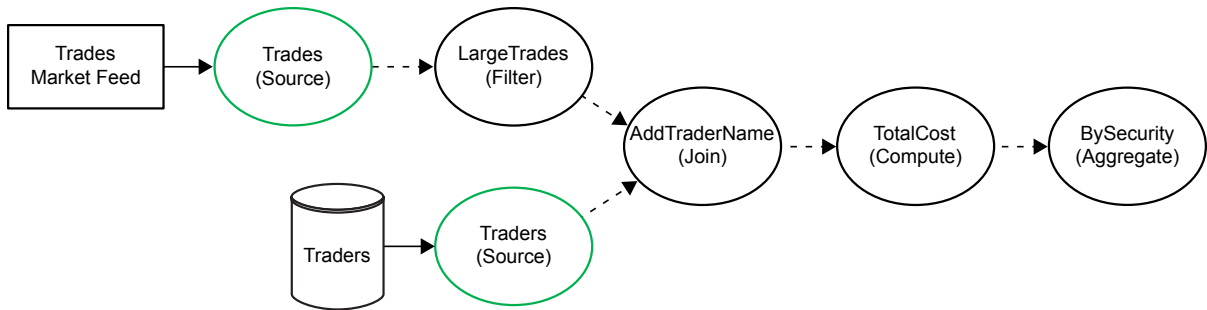
Various connectors and adapters are provided with SAS Event Stream Processing Engine, but you can write your own connector or adapter using the publish/subscribe API. Inside model definitions, you can define connector objects that can publish into source windows or that can subscribe to any window type.

Understanding Continuous Queries

Within a continuous query, windows can transform or analyze data, detect patterns, or perform computations. For information about the programming objects available with the SAS Event Stream Processing Engine that you use to implement windows, see [Chapter 2, “Programming with the C++ Modeling API,” on page 13](#).

Consider the following continuous query.

Figure 1.2 Continuous Query Diagram



In this continuous query, there are two source windows:

- the Trades window streams data about securities transactions from a trades market feed
- the Traders window streams data about who performs those transactions

After the source windows get data, the following occurs:

- 1 The Trades source window flows into the LargeTrades derived window, which filters out transactions that involve fewer than a defined number of shares.
- 2 LargeTrades and Traders flow into the join window named AddTraderName. This window matches filtered transactions with their associated traders.
- 3 Events from AddTraderName flow into the compute window named TotalCost, where the cost of the transaction is calculated.
- 4 Events are passed on to the aggregate window BySecurity, where they are placed into aggregate groups.

For C++ source code that implements a model that includes this query, see [Appendix 1, “Example: Implementation of the Trades Model,”](#) on page 227. For an example of the model rendered in the `esp_xml` modeling language, see [“Complete Example”](#) on page 62.

Specifically, 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.

Understanding Events

An event is a packet of data that is accessible as a collection of fields. Each event must have one or more fields 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/subscribe 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 helps throughput performance with minimal impact on latency. This assumes that event blocks are relatively small, around 512 bytes in length or less per event block, depending on the width of each event.

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.

Using the SAS Event Stream Processing Engine

After you install and configure SAS Event Stream Processing Engine, you program event stream processing applications to use it. Sample applications are available in the `src` directory of the installation. Each subdirectory contains the sample application's

source, a Makefile to build it, and in some cases sample data. Edit the Makefile to set options appropriately for your compiler and operating system. Instructions provided in the comments of each Makefile explain how to do this. A Readme file in the `src` directory explains each example and how to run it.

A utility called `dfespenv` in the `bin` directory can be used to set the environment variables that you need to run the compiled applications. Running `dfespenv` without any arguments displays usage instructions. You can add the commands to set the environment variables to your `.profile` or `.login` file when you want to make these settings permanent.

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 within an XML factory server.
- 2 Publish one or more event streams into the engine using the publish/subscribe API or by event injection.

- 3 Subscribe to relevant window event streams within continuous queries using the publish/subscribe API or by registering a callback function with the relevant window.

You can use the C++ Modeling API or the XML Modeling Layer to write an application. For more information about the C++ key modeling objects to use to write an application, see [Chapter 2, “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. This modeling layer does not provide all the capabilities that the C++ Modeling API provides. However, it does provide the essential modeling objects and connectors. For more information, see [Chapter 3, “Using the XML Modeling Layer,” on page 51](#).

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 data method of continuous queries

For more information about the publish/subscribe API, see [Chapter 8, “Using the Publish/Subscribe API,” on page 99](#).

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 [“Using Connectors” on page 124](#).

Adapters use the corresponding connector class to provide stand-alone executables that use the publish/subscribe API. Therefore, they can be networked. For more information, see [“Using Adapters” on page 155](#).

2

Programming with the C++ Modeling API

Using Expressions	14
Overview to the C++ Modeling API	16
Dictionary	17
dfESPengine	17
dfESPproject	18
dfESPeventdepot	21
dfESPcontquery	21
dfESPwindow_source	22
dfESPwindow_filter	24
dfESPwindow_copy	27
dfESPwindow_compute	29
dfESPwindow_union	33
dfESPwindow_aggregate	34
dfESPwindow_join	35
dfESPwindow_pattern	39
dfESPwindow_procedural	40
dfESPdatavar	41
dfESPschema	43
dfESPevent	44
dfESPeventblock	46
dfESPpersist	47

Using 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 employ 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 employ 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.

The SAS Event Stream Processing Engine API does not have a one-to-one mapping between the data types that it supports 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)

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). After you do, set the environment variable `DFESP_QKB` to the root path where they were installed. After that, you can include these functions in any of your SAS event stream processing expressions. These functions are typically used for cleaning up event fields in the non-key field calculation expressions in a compute window.

For an example, see [Appendix 3, “Example: Using Blue Fusion Functions,”](#) on page 241.

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 applications process space. An application could simply define and start an engine, which would make it an event stream processing server.

Alternatively, you could 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 instantiate an engine model inside of an XML factory server engine, see [Chapter 3, “Using the XML Modeling Layer,”](#) on page 51.

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 *-bbadevent.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`.

Example

```
// 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 pattern, 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 211](#).

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 213](#).

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 211](#).

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` function, 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

Example

The following code shows how to register the Boolean filter through a function. In this case, the function is named `booleanScalarFunction`.

```
// 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.

The following code shows how to register the Boolean filter for an expression.

```

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, because setting new policies can be performed only by 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 211](#).

schema

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

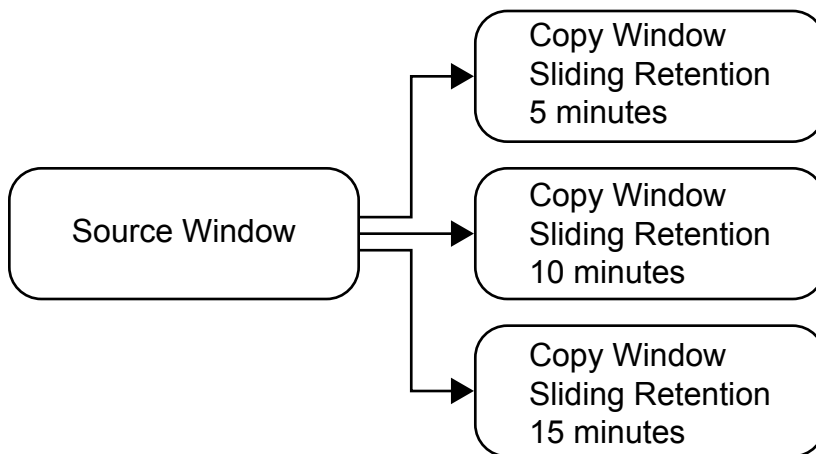
Details

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 213](#).

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 2.1 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 213](#).

dfESPwindow_compute

defines a compute window, which gets the key fields of its schema from its parent input window. Non-key fields are computed from the fields of the input event. Compute windows are projections of the input data using the window’s non-key field computation functions or expressions.

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 211](#).

schema

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

Example

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

The following example creates a compute window using a collection of function pointers. Assume that the input events have this schema:

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

The compute window then 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` have a prototype of type `dfESPscalar_func` that is defined in the header file `api/dfESPfuncptr.h`.

Here is the scalar function:

```
dfESPdatavar *priceBYquant(dfESPschema*is, dfESPevent *nep,
    dfESPevent *oep) {
    //
    // 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;
}
```

When specifying the schema for a compute window, the specified key fields are compared to the key fields of the input window. The data types must match. For example, for all keys, the kth key of the input schema must have the same data type as the kth key of the schema specified for the compute window.

Note: This check is not performed when the compute window is defined. This is because the input window can be specified at that point. The check is performed much later when the project that contains the continuous query that holds the compute window is started. This occurs when the graph is scanned and all connectivity-related checks are completed.

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.

The following example creates a compute window using field calculation expressions rather than a function.

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

The example 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.

```

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. This is why you can have 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 together that have 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 211](#).

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 have the same schema. The default value of the strict flag is `true`, which means that the key merge from each window must semantically merge cleanly. In that case, you cannot send an Insert event for the same key using two separate input windows of the union window.

When the strict flag is set to `false`, it loosens the union criteria by replacing all incoming Inserts with Upserts. All incoming Deletes are replaced with safe Deletes. In that case, deletes of a non-existent key fail without generating an error.

dfESPwindow_aggregate

specifies an aggregation window. An aggregation window is similar to a compute window in that non-key fields are computed. However, unlike compute windows, key fields are specified and not inherited from the input window.

Syntax

```
dfESPwindow_aggregate *windowID;
windowID=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 211](#).

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 81](#)

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 211](#).

Details

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

- left outer window
- right outer window
- inner 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.

The following example shows the 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.

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.

There are, however, 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. It has the aggregation 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 use 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. This eliminates all table scans when changes are made to the dimension table, and thus it performs 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 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.

See Also

[“Using Joins and Understanding Join Constraints” on page 203](#)

dfESPwindow_pattern

enables the detection of events of interest with a window pattern. A window pattern 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 211](#).

schema

schema associated with the window feeding the pattern window

See Also

[Chapter 4, “Creating Pattern Windows,” on page 67](#)

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 211](#).

schema

schema defined by dfESPstring

See Also

[Chapter 6, “Creating Procedural Windows,” on page 85](#)

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.

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

$\{i|u|p|d\}, \{n|s\}, f_1, f_2, \dots, f_n$ where

$\{i|u|p|d\}$ means Insert, Update, Upsert, and Delete respectively

$\{n|p\}$ means normal event or partial-update event

f_1, \dots, f_n are the n 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 204](#).

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.

Event blocks have a unique embedded transaction ID generated as they are absorbed into a continuous query. They also have a unique ID that the publisher can assign to it. 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

Details**Persisting an Engine**

Consider the following code:

```
// 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();
```

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. For more information, see [“Persist and Restore Model State” on page 220](#).

Restoring an Engine

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.

You can also restore state programmatically:

```
// 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 success = 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.
    //
}
dfESPengine::startProjects(); // Start all projects from restored state of engine.
```


3

Using the XML Modeling Layer

<i>XML Modeling Layer</i>	52
Overview to the XML Modeling Layer	52
High-Level Syntax of XML Models	52
Window Template	53
Window Where Nesting Defines Connectivity	55
<i>Using the XML Factory Server</i>	56
Overview to Using the XML Factory Server	56
Starting the XML Factory Server	56
Control Commands for the XML Factory Server	57
<i>Examples</i>	57
Creating a Project	57
Stopping a Project	58
Removing a Project	58
Events Publishing through the XML Factory Server	59
Combined Model-Event Processing for Short-lived Projects	59
Querying Engine Windows through the XML Factory Server	61
Querying the Engine Window Event Count	62
Complete Example	62

XML Modeling Layer

Overview to the XML Modeling Layer

The XML modeling layer is a higher level abstraction of the C++ Modeling API. This modeling layer enables someone without a background in programming to build event stream processor models. Although it provides the major modeling capabilities, it does not map one-to-one to the C++ modeling methods. Only expressions for field calculations, joins, filters, and so on, are supported.

Specifically, the following features that are available through the C++ Modeling API are not available through the XML modeling layer:

- The ability to preset the size of an index. In the C++ Modeling API, you can do this for all indices produced by a `dfESPeventdepot` or on a window-by-window basis (passed as an argument when the window is created).
- using engine persist and restore
- DS2 methods on procedural windows
- the capability to turn on and off propagation of redundant updates for a window

High-Level Syntax of XML Models

Here is the high-level syntax of XML models, including the engine, project, and continuous queries:

```
<engine>
  <projects>
    <project name='unique project name'
      [pubsub={'auto'|'manual'}]
      threads='10'>
      <window-templates>
        <window-template ...>
        </window-template>
        ...
      <!-- MORE WINDOW-TEMPLATE elements -->
```

```

    </window-templates>

    <contqueries>
      <contquery name='name of contquery'>
        <window ...>
          <!-- ARBITRARY nesting of windows -->
        </window>
        ...

        <window ...>
          <!-- ARBITRARY nesting of windows -->
        </window>
      </contquery>
      ...
    <!-- MORE CONTQUERY elements -->
  </contqueries>
</project>
<!-- MORE PROJECT elements... -->

</projects>
</engine>

```

Window Template

Here is the window-template element in detail:

```

<window-template name='name of window template'
  type={'source'|'filter'|'functional'|
    'aggregate'|'join'|'copy'|'compute'|'union'|
    'procedural'|'pattern'}
  [index={'empty','hash','rbtree'}] [trace={'true'|'false'}]>
  [ <schema>
    <fields>
      <field name='name of field' type={'int32'|'int64'|
        'double'|'money'|'date'|'stamp'} [id='true']/>
      ...
      <field name='name of field' type={'int32'|'int64'|
        'double'|'money'|'date'|'stamp'} [id='true']/>
    </fields>
  </schema>]

  [ <connectors>
    <connector class={'fs', 'db', 'smtp',
      data set, jms, mqueue}
    <properties>

```

```

        <property name='property name'>property
            value</property>
        ...
        <property name='property name'>property
            value</property>
    </properties>
</connector>
...
<!-- MORE CONNECTOR elements -->
</connectors> ]

[ <inputs>
    <input source='ref to another window-template name'>
        <functions>
            <function field='field name'>function body
                </function>
            ...
            <function field='field name'>function body
                </function>
        </functions>
    </input>
    ...
    <!-- MORE INPUT elements -->
</inputs>]

[ <outputs>
    <field name='field name' type={'int32'|'int64'|
        'double'|'money'|'date'|'stamp'}
        [id='true']/>
    ...
    <field name='field name' type={'int32'|'int64'|
        'double'|'money'|'date'|'stamp'}
        [id='true']/>
</outputs>]

[ <expression>expression body</expression>]

[ <join type={'leftouter'|'rightouter'|'inner'}>
    <conditions>l_field=r_field,...,l_field=r_field
    </conditions>
    <left source='ref to another window-template name'>
        <outputs>
            <field name='field name' type={'int32'|
                'int64'|'double'|'money'|'date'|
                'stamp'}
                [id='true']/>
            ...
            <field name='field name' type={'int32'|

```

```

        'int64'|'double'|'money'|'date'|
        'stamp'}
      [id='true']/>
    </outputs>
  </left>
  <right source='ref to another window-template name'>
    <outputs>
      <field name='field name' type={'int32'|
        'int64'|'double'|'money'|'date'|
        'stamp'}
      [id='true']/>
      ...
      <field name='field name' type={'int32'|
        'int64'|'double'|'money'|'date'|
        'stamp'}
      [id='true']/>
    </outputs>
  </right>
</join>

</window-template>

```

Window Where Nesting Defines Connectivity

Here is the window element in detail where nesting defines connectivity:

```

<window name='name of window' [pubsub='true']
  [trace={'true'|'false'}]>
  <window name='name of window' [pubsub='true']
    [trace={'true'|'false'}]>
    </window>
    ...
  <window name='name of window' [pubsub='true']
    [trace={'true'|'false'}]>
    </window>
  </window>
</window>

```

Using the XML Factory Server

Overview to Using the XML Factory Server

The XML factory server, `dfesp_xml_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.

In order to interact with the XML factory server, you need a tool to read and write to the server port defined in the engine model. Alternatively, you can specify the port as a command line argument when you execute `dfesp_xml_server`. This factory server control client needs to be written by event stream processor application developers.

Starting the XML Factory Server

There are two ways to start the server:

- Specify an `<esp-server port='...' />` element in the XML model configuration file. This is different from the engine port that is used for publish/subscribe operations. For example, suppose the XML model file contains these lines:

```
<engine port='9980' dateformat='%d/%b/%Y:%H:%M:%S'>
  <esp-server port='9990' />
  ...
</engine>
```

In this case, port 9990 is used by the XML factory server. If the port is not specified within `<esp-server> ... </esp-server>`, then it defaults to the engine publish/subscribe port +1, which in this case would be 9981.

Run the following command: `$DFESP_HOME/bin/dfesp_xml_server-model file://model_filename.`

- Specify the `-server port` option on the command line without the quotation marks. Again, if no argument is provided, the server port defaults to the engine publish/subscribe port + 1.

Run the following command: `$DFESP_HOME/bin/dfesp_xml_server-model file://model_filename -server 33333.`

Control Commands for the XML Factory Server

You can execute the XML factory server with a complete engine mode definition file using the `file://` syntax shown previously. The engine model can contain zero or more defined projects. In many cases the XML factory server is used to execute static models, but it can also be used to execute more dynamic models. For static models, the engine definition would contain one or more projects. The server socket interface would not be used for creating and removing projects. For dynamic models, the engine definition could contain zero or more projects. The server socket interface would be used for creating or removing projects (or both) from the running engine.

Examples

Creating a Project

Here is an example of creating a project named “events”:

```
<project action='load' name='events' pubsub='auto'>
  <window-templates>
    <window-template name='event_t' type='source'>
      <schema>
        <fields>
          <field name='id' type='string' id='true' />
          <field name='user' type='string' />
          <field name='date' type='date' />
        </fields>
      </schema>
    </window-template>
  </window-templates>
</project>
```

```
</schema/>
</window-template>
</window-templates>
<contqueries>
  <contquery name='events'>
    <window name='events' window-template='event_t' trace='true'/>
  </contquery>
</contqueries>
</project>
```

The successful response from the XML factory server is as follows:

```
<response action='load' name='events' pubsub='auto' elapsed='2 ms'>
  <message>project load events succeeded</message>
</response>
```

Stopping a Project

The following code stops the project named Events:

```
<project name='events' action='stop'/>
```

The successful response from the XML factory server is as follows:

```
<response action=stop name='events' elapsed='994 ms'>
  <message>project events stopped</message>
</response>
```

Removing a Project

The following code removes the project named Events from the XML factory server engine definition:

```
<project name='events' action='remove'/>
```

The successful response from the XML factory server is as follows:

```
<response action=remove name='events' elapsed='1005 ms'>
  <message>project removal succeeded</message>
</response>
```

Events Publishing through the XML Factory Server

Inject events into a running event stream processing engine: The XML factory server supports Insert, Update, Upsert, and Delete opcodes.

Here is an example of injecting two events into the project that was created in the previous example:

```
<inject>
  <events project='events' contquery='events' window='events'>
    <event opcode='insert'>
      <value name='id'>10</value>
      <value name='user'>Eric</value>
      <value name='date'>1/aug/2013:09:00:00</value>
    </event>
  </events>
</inject>
```

After events are successfully injected to the server, the XML factory server writes the response in XML:

```
<response elapsed='0 ms'>
  <message>2 events injected into events:events:events</message>
</response>
```

Combined Model-Event Processing for Short-lived Projects

Combined processing enables you to send over the model or project. Then you can inject data as specified in event-stream element into the model, and get the result back in the response from the XML factory server. The project lives for only the request. It is removed after the request. The model configuration file is not required in this case.

Start the server as follows : `dfesp_xml_server -server port`

Here is the XML example to send to the XML server:

```
<stream>
  <project>
    <window-templates>
      <window-template name='event_t' type='source'>
```

```

    <schema>
      <fields>
        <field name='id' type='string' id='true' />
        <field name='user' type='string' />
      </fields>
    </schema>
    <window-template name='users_t' type='aggregate'>
      <schema>user*:string,numEvents:int64</schema>
</output>
    <field>ESP_aCount()</field>
  </output>
</window-template>
</window-templates>
<contqueries>
  <contquery name='events'>
    <window name='events' window-template='event_t'>
      <window name='users' window-template='users_t' />
    </window>
  </contquery>
</contqueries>
</project>
<event-stream contquery='events' window='events'>
  <events>
    <event opcode='insert'>
      <value name='id'>10</value>
      <value name='user'>John</value>
    </event>
    <event opcode='insert'>
      <value name='id'>20</value>
      <value name='user'>Eric</value>
    </event>
  </events>
</event-stream>
<results>
  <window contquery='events' name='users' />
</results>
</stream>

```

The response from the XML server is as follows:

```

<response elapsed='1006ms'>
  <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'>Eric</value>
      <value column='numEvents'>1</value>
    </data>
    <data>
      <value column='user'>John</value>
      <value column='numEvents'>1</value>
    </data>
  </results>
</events>
</response>

```

Querying Engine Windows through the XML Factory Server

You can query a window through the XML factory server. The response from the XML factory server contains the schema of the window and the data currently in the window. Here is an example of a query:

```
<query project='events' contquery='events' window='events' />
```

Here is the response to that query:

```

<response contquery='events' project='reference' window='events' elapsed='0 ms'>
  <window>
    <columns>
      <column label='id' name='id' sqltype='12' type='varchar' />
      <column label='user' name='user' sqltype='12' type='varchar' />
      <column label='date' name='date' sqltype='91' type='date' />
    </columns>
    <results>
      <data>
        <value column='date'>01/Aug/2013:13:00:00</value>
        <value column='id'>20</value>
        <value column='user'>scott</value>
      </data>
      <data>
        <value column='date'>01/Aug/2013:12:00:00</value>
        <value column='id'>10</value>
        <value column='user'>scott</value>
      </data>
    </results>
  </window>
</response>

```

Querying the Engine Window Event Count

You can issue a query to ask the XML factory server to provide the number of events that are in a window:

```
<count project='events' contquery='events' window='events' />
```

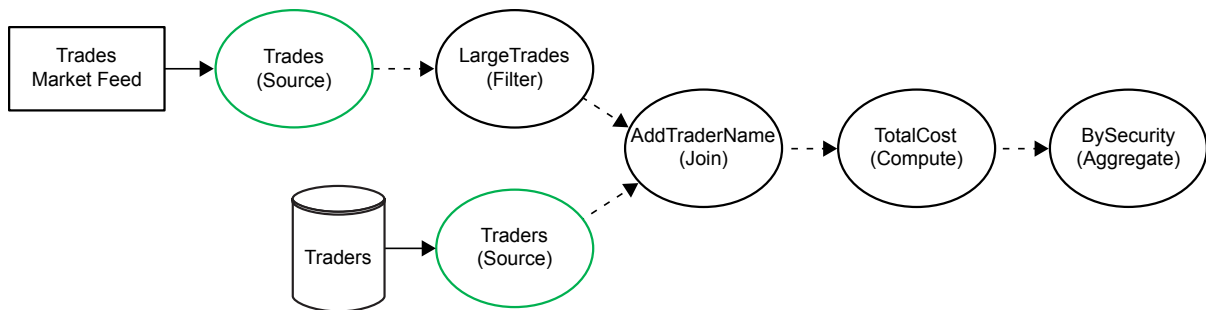
The response would appear as follows:

```
<response contquery='events' project='events' window='events' elapsed='0 ms'>
  <count project='reference' contquery='events' window='events'>2</count>
</response>
```

Complete Example

Recall the continuous query described in “Understanding Continuous Queries” on page 5.

Figure 3.1 Continuous Query Diagram



The following code renders the model containing that continuous query in the `esp_xml` modeling language:

```
<engine port='55555' dateformat='%d/%b/%Y:%H:%M:%S' trace='false'>
  <projects>
    <project name='project_01' pubsub='auto' threads='10'>
      <window-templates>
        <window-template name='Trades_t' type='source'>
          <schema>
            <fields>
              <field name='tradeID' type='string'
                id='true' />
            </fields>
          </schema>
        </window-template>
      </window-templates>
    </project>
  </projects>
</engine>
```

```

        <field name='security' type='string' />
        <field name='quantity' type='int32' />
        <field name='price' type='double' />
        <field name='traderID' type='int64' />
        <field name='time' type='stamp' />
    </fields>
</schema>
</window-template>

<window-template name='Traders_t' type='source'>
    <schema>
        <fields>
            <field name='ID' type='int64'
                id='true' />
            <field name='name' type='string' />
        </fields>
    </schema>
</window-template>

<window-template name='LargeTrades_t' type=
    'filter'><expression>quantity >=
    100</expression>
</window-template>

<window-template name='AddTraderName_t'
    type='join'>
    <join type="leftouter">
        <conditions>l_traderID==r_ID
        </conditions>
        <left source="Trades">
            <output>
                <field name='security'
                    type='string' />
                <field name='quantity'
                    type='int32' />
                <field name='price'
                    type='double' />
                <field name='traderID'
                    type='int64' />
                <field name='time'
                    type='stamp' />
            </output>
        </left>
        <right source='Traders'>
            <output>
                <field name='name'
                    type='string' />

```

```

        </output>
    </right>
</join>
</window-template>

<window-template name="TotalCost_t" type="compute">
    <schema>
        <!-- These are the non-key fields -->
        <fields>
            <field name='tradeID' type='string'
                id='true' />
            <field name='security' type='string' />
            <field name='quantity' type='int32' />
            <field name='price' type='double' />
            <field name='totalCost' type='double' />
            <field name='traderID' type='int64' />
            <field name='time' type='stamp' />
            <field name='name' type='string' />
        </fields>
    </schema>
    <!-- These are how the non-key fields are
    computed-->
    <output>
        <field name='security'>security</field>
        <field name='quantity'>quantity</field>
        <field name='price'>price</field>
        <field name='totalCost'>price*quantity
            </field>
        <field name='traderID'>traderID</field>
        <field name='time'>time</field>
        <field name='name'>name</field>
    </output>
</window-template>

```

```

<window-template name='BySecurity_t'
    type='aggregate'>
    <schema>
        <fields>
            <field name='security' type='string'
                id='true' />
            <field name='quantityTotal'
                type='double' />
            <field name='costTotal'
                type='double' />
        </fields>
    </schema>

```



```

        <output>
            <field>ESP_aSum(quantity)</field>
            <field>ESP_aSum(totalCost)</field>
        </output>
    </window-template>

</window-templates>

<contqueries>
    <contquery name='cq_01'>

<!-- Traders -> AddTraderName -->
        <window name='Traders' window-
            template='Traders_t' trace='true'>
            <window name="AddTraderName"
                window-template=
                    'AddTraderName_t'
                trace='true' />
            </window>
<!-- Trades -> LargeTrades -> AddTraderName
-> TotalCost ->
BySecurity -->
        <window name='Trades' window-template=
            'Trades_t' trace='true'>
            <window name='LargeTrades' window-
                template='LargeTrades_t'
                trace='true'>
            <window name="AddTraderName"
                window-template='AddTraderName_t'
                trace='true'>
            <window name='TotalCost'
                window-template='TotalCost_t'
                trace='true'>
            <window name='BySecurity'
                window-template='BySecurity_t'
                trace='true'>
            </window>
        </window>
    </window>
    </window>
    </contquery>
</contqueries>
</project>
</projects>
</engine>

```


4

Creating Pattern Windows

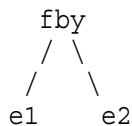
<i>Overview of Pattern Windows</i>	67
<i>State Definitions for Operator Trees</i>	70
<i>Restrictions on Patterns</i>	73
<i>Example: Simple Pattern Window</i>	75

Overview of Pattern Windows

A pattern is an algebraic expression using the logical operators AND, OR, NOT, 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 an example:

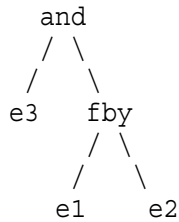
```
expression of interest e1: (a==10)
expression of interest e2: (b==5 and cost>100.00)
pattern: e1 fby e2
pattern tree view:
```



Here is a 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:
```



When you create a pattern window, you declare a list of events of interest (EOVs) and then connect those events into an expression that use the same logical operators. Many patterns are temporal. For example, you can have a combination of EOVs 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 EOVs by providing the following:

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

Suppose that you have 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 you have two EOVs 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 EOVs, *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 both matching events have the same value for the field *buy*.

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 EOVs on either side of the operator and for the first matching EOV 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 1 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 sells 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.

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 EOVs. Given that EO refers to an event of interest or operator tree (EOV|OPT):

not EOV

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

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 EOV

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

notoccur OPT

this is not allowed

EO or EO

an event 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 FALSE. 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

an event 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 followed by 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 to TRUE and FIXED.

This apply algorithm is driven to seek the minimal length sequence of events that completes an FBY pattern.

For example, to detect a, ..., b, where ... can be any sequence:

```
(a fby b)
```

To detect a, ..., b: but there can be no c between a and b:

```
( a fby ( (notoccur c) and b ) )
```

To detect a, X, b: when X cannot be c:

```
( a fby (not c) ) fby (not (not b))
```

And to detect a, ..., b, ..., c, ..., d : but k does not occur anywhere in the sequence:

```
( ( (a fby b) fby (c fby d) ) and (notoccur k) )
```

To detect an FBY b with no occurrences of c in the sequence:

```
a fby (notoccur(c) and b)
```

To detect an FBY b directly, with nothing between a and b:

```
(not not a) fby (not not b)
```

To detect a ... b ... d, with no occurrences of c between a and b:

```
a fby (b fby ((notoccur c) and d))
```

And to detect a ... b ... d, with no occurrences of c anywhere:


```
(notoccur c) and (a fby (b fby d))
```

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);
```

Example: Simple Pattern Window

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 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,
                                dfESPindextypes::pi_RBTREE,
                                dfESPstring("ID*:int64,ID1:
                                             int32,ID2:int32"));

// 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_01 --> 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

5

Creating Aggregate Windows

<i>Overview to Aggregate Windows</i>	81
<i>Flow of Operations</i>	82
<i>Aggregate Functions</i>	83

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 input events have a schema specified as the following:

```
"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. The non-key fields, in this example `totalQuant` and `maxPrice`, must have either aggregate field calculation functions (written in C++) or expressions that are registered to the aggregate window. 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, \mathfrak{E} arrives and the appropriate group is found, called \mathfrak{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 \mathfrak{E} is merged into the group \mathfrak{G} . The key of the output event is formed from the group-by fields of \mathfrak{G} .
- 3** Each non-key field of the output schema is computed by calling an aggregate function with the group \mathfrak{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.

Aggregate Functions

The following aggregate functions are available for aggregate window field calculation expressions:

Aggregate Function	Returns
<code>ESP_aCount()</code>	number of events in the group
<code>ESP_aGUI()</code>	a unique identifier
<code>ESP_aSum(fieldname)</code>	sum of the group
<code>ESP_aMin(fieldname)</code>	minimum of the group
<code>ESP_aMax(fieldname)</code>	maximum of the group
<code>ESP_aAve(fieldname)</code>	average of the group
<code>ESP_aWAvge(weight_fieldname, payload_fieldname, fieldname)</code>	weighted group average
<code>ESP_aLast(fieldname)</code>	field from the last record that affected the group
<code>ESP_aLastNonDelete(fieldname)</code>	value of <code>aLast()</code> for non-delete
<code>ESP_aLastOpcodes(opcode)</code>	the opcode of the last record to affect the group
<code>ESP_aCountOpcodes(opcode)</code>	count of the number of events matching opcode for group
<code>ESP_aFirst(fieldname)</code>	the first event added to the group
<code>ESP_aStd(fieldname)</code>	standard deviation of the group

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);
```

It can be found 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 Aggregation Functions” on page 215](#).

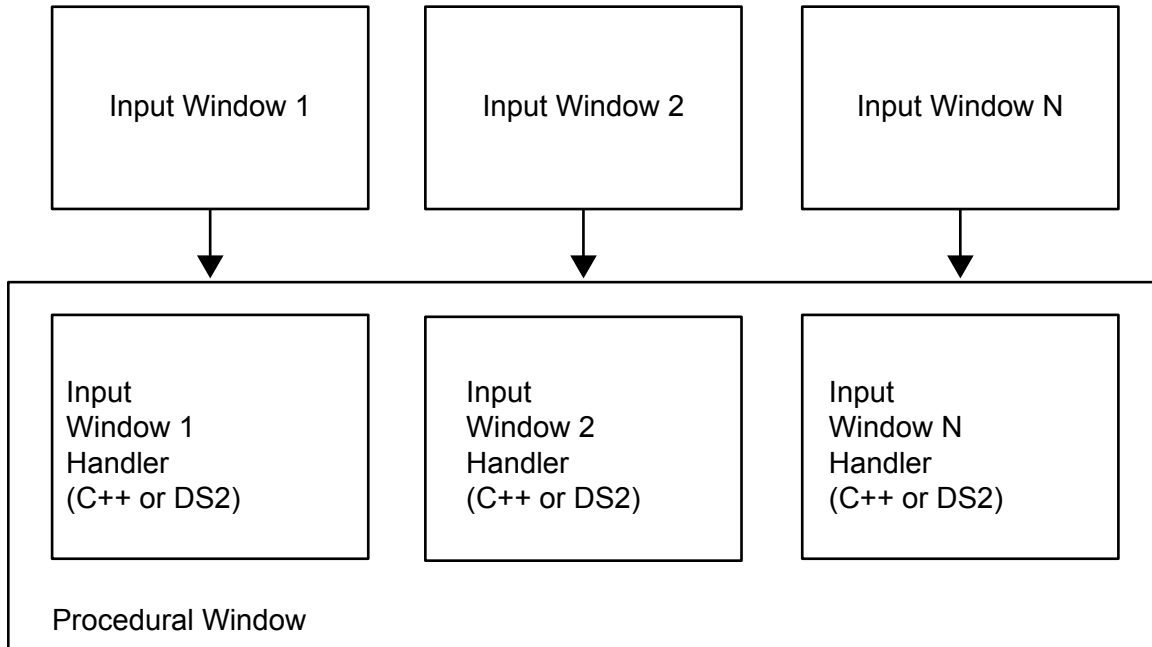
6

Creating Procedural Windows

- Overview to Procedural Windows* 85
- C++ Window Handlers* 86
- DS2 Window Handlers* 89
 - Overview of DS2 Window Handlers 89
 - General Structure of a DS2 Input Handler 89
 - Examples 90
 - Event Stream Processor to DS2 Data Type Mappings and Conversions 92

Overview to Procedural Windows

You can write procedural windows in C++ or DS2 (DataStep 2). 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 6.1 *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 DataStep 2 programming language (DS2), 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.

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

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 in the DS2 program. 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

Here is a procedural window with one input window that computes some new fields:

```

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, but 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 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 out events that have an even ID. It produces two identical events (with different keys) for those events that have 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

Event Stream Processor Data Type	DS2 Data Type
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

7

Visualizing Event Streams

<i>Overview to Event Visualization</i>	95
<i>Using Streamviewer</i>	95
<i>Using SAS/GRAPH</i>	97

Overview to Event Visualization

The Streamviewer tool provided with SAS Event Stream Processing Engine enables you to subscribe to engine's event streams. You can use this tool, which is named `dfesp_streamviewer` 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/src` 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\bin\graph_realtime` directory. In order to use it, you must separately purchase SAS/GRAPH.

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 URL_to_ESP_server
-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 URL_to_ESP_server
-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.

8

Using the Publish/Subscribe API

<i>Overview to the API</i>	99
<i>Understanding Publish/Subscribe API Versioning</i>	100
<i>Using Callback Functions</i>	101
<i>The API from the Engine's Perspective</i>	102
<i>The API from the Client's Perspective</i>	103
<i>Functions for the Publish/Subscribe API</i>	105
<i>Using the Java Publish/Subscribe API</i>	117
Overview to the Java Publish/Subscribe API	117
Using High Level Publish/Subscribe Methods	119
Using User-supplied Callback Functions	120

Overview to the API

The SAS Event Stream Processing Engine provides publish/subscribe application programming interfaces (APIs) for C and 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 currently supported on all supported hardware and operating system types as well as Microsoft 32-bit Windows® (which is publish/subscribe or client only). This level of support enables cross-platform usage. For example, you can subscribe to event streams in an SAS Event Stream Processing Engine running on Solaris SPARC. Those streams can be visually displayed on a Windows desktop, perhaps using SAS/GRAPH.

You can also subscribe to an event stream so that it can be loaded into a database for persistence. More likely you would use the event stream processor database adapter or connector. Connectors are in process classes that publish and subscribe. Adapters are stand-alone executables that publish and subscribe. Both use the publish/subscribe API.

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.

For examples of a subscriber and a publisher application, see [Appendix 2, “Example: Subscriber and Publisher Applications,”](#) on page 231.

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 Callback Functions

In addition to the publish/subscribe API, an SAS Event Stream Processing Engine application written using the C++ Modeling API can register a callback function with a window. This is done so that the application thread subscribes to that window while bypassing the publish/subscribe API. This is done by defining the call back function as described here:

```
void winSubscribe_function(dfESPschema *os, dfESPEventblockPtr ob,
    void *ctx) {

    // do something with the eventblock

}
```

Then you can register the callback function with the SAS Event Stream Processing Engine window of interest as follows:

```
windowPtr->addSubscriberCallback(winSubscribe_function);
```

Similarly, an SAS Event Stream Processing Engine application can bypass the publish/subscribe API and publish events into SAS Event Stream Processing Engine source windows using the `project injectData()` member function as follows:

```
projectPtr->injectData(contQueryPtr, sourceWindowptr, eventBlock);
```

Note: A demonstration of how to use `injectData()` and `addSubscriberCallback()` are in the Application Building section.

These methods give the SAS Event Stream Processing Engine application low-level access to publishing and subscribing. The publish/subscribe API enables other applications and clients to publish and subscribe to an SAS Event Stream Processing Engine application either on the same machine or from a system on the network. When using subscriber clients, the SAS Event Stream Processing Engine project windows need to be enabled for subscription.

The 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 you would 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 example opens a server listener socket on port 33335 to allow client subscribers and publishers to connect to the SAS Event Stream Processing Engine application or server for publish/subscribe services. Once 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 where you need to override ephemeral ports for a specific port (for security purposes), then that can be done by providing `project->setPubSub` with a second parameter, which 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, windows can be enabled manually by specifying `ps_MANUAL`. For non-trivial projects, the specific windows of interest should be enabled 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 `ps_MANUAL` was used in `project->setPubSub()` to specify manual enablement of window subscribes, then `enableWindowSubs()` is subsequently used 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 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 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	<code>level</code> the logging level
	<code>logConfigPath</code> the full pathname to the log configuration file

int C_dfESPpubsubInit(C_dfESPLoggingLevel level, const char *logConfigPath)

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 <code>C_dfESPpublisherStart()</code> . 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	<p>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.</p> <p>serverURL format "dfESP://host:port/project/contquery/window?snapshot=true false"</p> <p>callbackFunction a pointer to a user-defined function for handling received event blocks. This function must call <code>C_dfESPeventblock_destroy()</code> to free the event block.</p> <p>errorCallbackFunction either NULL or a user-defined function pointer for handling subscription service failures</p> <p>ctx optional context pointer for parsing state into this call</p>
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 <code>C_dfESPsubscriberStart()</code> . Additional required parameter: filename of this subscriber's guaranteed delivery configuration file.
------------	--

int C_dfESPpubsubConnect(clientObjPtr client)

Parameter	<code>client</code> pointer to a client object returned by <code>C_dfESPsubscriberStart()</code> or <code>C_dfESPpublisherStart()</code> or <code>C_dfESPGDsubscriberStart()</code> or <code>C_dfESPGDpublisherStart()</code>
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	<code>client</code> pointer to a client object returned by <code>C_dfESPsubscriberStart()</code> or <code>C_dfESPpublisherStart()</code> or <code>C_dfESPGDsubscriberStart()</code> or <code>C_dfESPGDpublisherStart()</code> <code>block</code> 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	<p><code>client</code> pointer to a client object returned by <code>C_dfESPsubscriberStart()</code> or <code>C_dfESPpublisherStart()</code> or <code>C_dfESPGDsubscriberStart()</code> or <code>C_dfESPGDpublisherStart()</code></p> <p><code>block</code> set to 1 to wait for all queued events to be processed, else 0</p>
Return values	<p>1 success</p> <p>0 failure — error logged to the SAS Event Stream Processing Engine log</p>

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

int C_dfESPpublisherInject(clientObjPtr client, C_dfESPEventblock eventBlock)

Parameters	<p><code>client</code> pointer to a client object returned by <code>C_dfESPpublisherStart()</code> or <code>C_dfESPGDsubscriberStart()</code></p> <p><code>eventBlock</code> 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.</p>
Return values	<p>1 success</p> <p>0 failure — error logged to the SAS Event Stream Processing Engine log</p>

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)

Parameters	<i>queryURL</i> 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* are as follows:

"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

<code>"dfESP://host:port?get=windows"</code>	returns names of windows in currently running projects
<code>"dfESP://host:port?get=windows_pubsubonly"</code>	returns names of publish/subscribe enabled windows in currently running projects
<code>"dfESP://host:port/project?get=windows"</code>	returns names of windows in the specified project, if running
<code>"dfESP://host:port/project?get=windows_pubsubonly"</code>	returns names of publish/subscribe-enabled windows in the specified project, if running
<code>"dfESP://host:port/project?get=queries"</code>	returns names of continuous queries in the specified project, if running
<code>"dfESP://host:port/project?get=queries_pubsubonly"</code>	returns names of continuous queries containing publish/subscribe-enabled windows in the specified project, if running
<code>dfESP://host:port/project/contquery?get=windows"</code>	returns names of windows in the specified continuous query and project, if running

<code>dfESP://host:port/project/contquery? get=windows_pubsubonly"</code>	returns names of publish/subscribe-enabled windows in the specified continuous query and project, if running
<code>dfESP://host:port/project/contquery/window? get=schema"</code>	returns a single string that is the serialized version of the window schema
<code>dfESP://host:port/project/contquery/window? get=edges"</code>	returns the names of all the window's edges

C_dfESPstringV C_dfESPpubsubGetModel(char *queryURL)

Parameters	<p><i>queryURL</i> string representing the query to be posted to the SAS Event Stream Processing Engine.</p> <p>Supported formats of <i>queryURL</i> are as follows:</p> <ul style="list-style-type: none"> ■ "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	<p>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 "<i>project/query/window: edge1, edge2, ...</i>". The end of the list is denoted by an empty string. The caller is responsible for freeing the vector by calling <code>C_dfESPstringV_free()</code>.</p>

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 value 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_dfESPpubsubSetPubsubLib(C_dfESPpsLib psLib)

Parameters

psLib

enum representing the client/server transport

Supported values:

- 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"
}
```

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";
    }
}
```

- 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:

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib psLib)

Return value	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_dfESPdpgublisherStart()
C_dfESPdpgublisherGetID()
C_dfESPdpsubscriberStart()
C_dfESPdpsubscriberAck()
C_dfESPpubsubSetBufferSize()
```

C_dfESPdpsubscriberAck()

Parameters	Triggers an acknowledged. <ul style="list-style-type: none"> ■ Parameter 1: client object pointer. ■ Parameter 2: event block pointer. The event block must not be freed before this function returns.
Return value	<ul style="list-style-type: none"> ■ 1 = success ■ 0 = failure

C_dfESPGDpublisherCB_func()

Parameters	Signature of the new publisher callback function passed to C_dfESPGDpublisherStart() <ul style="list-style-type: none">■ 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()
------------	---

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	<i>client</i> pointer to a client object returned by C_dfESPsubscriberStart(), C_dfESPpublisherStart(), C_dfESPGDsubscriberStart(), or C_dfESPGDpublisherStart() <i>mbytes</i> the read and write buffer size, in units of 1MB
------------	---

Return value	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

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++ 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`. Navigate to `$DFESP_HOME/doc/html` to access the documentation.

Method	Description
<code>boolean init(Level level)</code>	Initialize publish/subscribe services
<code>dfESPclient publisherStart(String serverURL, clientCallbacks userCallbacks, Object ctx)</code>	Start a publisher
<code>dfESPclient subscriberStart(String serverURL, clientCallbacks userCallbacks, Object ctx)</code>	Start a subscriber
<code>boolean connect(dfESPclient client)</code>	Connect to the Event Stream Processor application or server
<code>boolean publisherInject((dfESPclient client, dfESPEventblock eventblock)</code>	Publish event blocks
<code>ArrayList< String > queryMeta (String queryURL)</code>	Query model metadata
<code>ArrayList< String > getModel(String queryURL)</code>	Query model windows and their edges
<code>boolean disconnect (dfESPclient client, boolean block)</code>	Disconnect from the event stream processor
<code>boolean stop (dfESPclient client, boolean block)</code>	Stop a subscriber or publisher
<code>void shutdown ()</code>	Shutdown publish/subscribe services
<code>boolean setBufferSize(dfESPclient client, int mbytes)</code>	Change the default socket read and write buffer size
<code>dfESPclient GDsubscriberStart (String serverURL, clientCallbacks userCallbacks, Object ctx, String configFile)</code>	Start a guaranteed delivery subscriber

Method	Description
<code>dfESPclient GdpublisherStart (String serverURL, clientCallbacks userCallbacks, Object ctx, String configFile)</code>	Start a guaranteed delivery publisher
<code>long GdpublisherGetID()</code>	Get a sequentially unique ID to write to an event block to be published using guaranteed delivery
<code>boolean GDsubscriberAck(dfESPclient client, dfESPEventblock eventblock)</code>	Trigger a guaranteed delivery acknowledgment
<code>boolean persistModel(String hostportURL, String persistPath)</code>	Instruct a running engine to persist its current state to disk

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

9

Using Connectors and Adapters

<i>Using Connectors</i>	124
Overview to Connectors	124
Activating Optional Plug-ins	126
Using File and Socket Connectors	127
File and Socket Connector Publisher Blocking Rules	129
XML File and Socket Connector Data Format	129
JSON File and Socket Connector Data Format	130
Syslog File and Socket Connector Notes	131
Using Database Connectors	131
Subscriber Event Stream Processor to SQL	
Data Type Mappings	134
Publisher SQL to Event Stream Processor Data	
Type Mappings	135
SMTP Subscribe Connector	136
IBM WebSphere MQ Connector	137
Tervela Data Fabric Connector	140
Solace Systems Connector	145
Tibco Rendezvous (RV) Connector	149
User-Written Connectors	152
Integrating a User-Written Connector	155
<i>Using Adapters</i>	155
Overview to Adapters	155
File and Socket Adapter	156
Database Adapter	158
SMTP Subscriber Adapter	160

Event Stream Processor to Event Stream	
Processing Engine Adapter	161
SAS Data Set Subscriber Adapter	161
Java Message Service (JMS) Adapter	163
IBM WebSphere MQ Adapter	165
Tervela Data Fabric Adapter	167
Solace Systems Adapter	169
Tibco Rendezvous (RV) Adapter	171
PI Publisher Adapter	173

Using Connectors

Overview to Connectors

Connectors are available in libraries located at `$DFESP_HOME/plugins_directory`. They 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 from the specified source
 - inject those events 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.

Connectors do not simultaneously publish and subscribe.

All connector classes are derived from a base connector class that is included in a connector library. The 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 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 [“User-Written Connectors” on page 152](#).

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`

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"`
- `"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”
- “tva”
- “pi”

Using 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)

The file or socket nature of the connector is specified by the form of the configured `fspath`. 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 `fspath`, 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 `fspath` 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 9.1 Required Parameters for File and Socket Connectors

Parameter	Description
type	Specifies whether to publish or subscribe
fstype	binary/csv/xml/json/syslog
fsname	Specifies the input file for publishers, output file for subscribers, or socket connection in the form of <i>host: port</i> . Leave <i>host</i> blank to implement a server instead of a client.

Table 9.2 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.
periodicity	Specifies the interval at which the subscriber output file is closed and a new output file opened. When configured, all output filenames have a timestamp appended for when that file was opened. This parameter does not apply to socket connectors.
maxfilesize	Specifies the maximum size in bytes of the subscriber output file. When reached, a new output file is opened. When configured, all output filenames have a timestamp appended. This parameter does not apply to socket connectors.

Table 9.3 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.

Parameter	Description
<code>transactional</code>	Sets the event block type to transactional. The default value is normal.
<code>growinginputfile</code>	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.
<code>rate</code>	Controls the rate at which event blocks are injected. Specified in events per seconds.
<code>maxevents</code>	Specifies the maximum number of events to publish.
<code>prebuffer</code>	Controls whether event blocks are buffered to an event block vector before doing any injects. The default value is false. Not valid with <code>growinginputfile</code> or for a socket connector

File and Socket Connector Publisher Blocking Rules

Input data used by a file and socket connector publisher can contain optional transaction delimiters. 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, event blocks have a size equal to the `blocksize` parameter. If the `blocksize` parameter is not configured, the default blocksize 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 the XML output XML:

```
<?xml version="1.0" encoding="utf-8"?>
```

The same header is support by the XML file and socket publisher.

The following tags are valid:

- “project”
- “contquery”
- “window”
- “transaction”
- “event”
- “opcode”

In addition, any tags corresponding to event data field names are valid when contained within an event. All of these tags are required except “transaction”. For more information, see [“File and Socket Connector Publisher Blocking Rules” on page 129](#).

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: “i”, “u”, “p”, “d”, and “s”, for Insert, Update, Upsert, Delete, and Safedelele respectively. 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 have value = NULL (or partial-update) if missing, or if the field contains 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, where the expected format is as follows:

```
month day hours : minutes : seconds hostname process : message
```

From each line, the connector extracts *date*, *hostname*, *process*, and *message*. The resulting event requires the following schema:

```
ID*:int32_u date:date_hostname:string.process:string.message:string
```

The connector reports an error when the schema of the target window does not match.

Using Database Connectors

A database connector that supports both publish and subscribe operations is available. It uses the DataDirect ODBC driver. Currently, it is certified for Oracle, MySQL, and DB2. It 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 9.4 Required Parameters for Subscriber Database Connectors

Parameter	Description
type	Specifies to subscribe.
connectstring	Specifies the database DSN and user credentials in the format "DSN= <i>dsn</i> ;uid= <i>userid</i> ;pwd= <i>password</i> ;"
tablename	Specifies the target table name.

Table 9.5 Required Parameters for Publisher Database Connectors

Parameter	Description
type	Specifies to publish.
connectstring	Specifies the database DSN and user credentials in the format "DSN= <i>dsn</i> ;uid= <i>userid</i> ;pwd= <i>password</i> ;"
selectstatement	Specifies the SQL statement executed on the source database.

Table 9.6 Optional Parameters for Subscriber Database Connectors

Parameter	Description
snapshot	Disables the sending of snapshot data. The default value is enabled.

Table 9.7 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
SP_MONEY	SQL_DOUBLE (converted to ESP_DOUBLE), SQL_FLOAT, SQL_REAL, SQL_NUMERIC (converted to SQL_NUMERIC), SQL_DECIMAL (converted to SQL_NUMERIC)

Subscriber Event Stream Processor Data Type	SQL Data Type
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

Publisher SQL Data Type	Event Stream Processor Data Types
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
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_DOUBLE, ESP_MONEY (converted from SQL_NUMERIC)
SQL_BINARY	ESP_UTF8STR
SQL_VARBINARY	ESP_UTF8STR
SQL_LONGVARBINARY	ESP_UTF8STR

SMTP Subscribe Connector

A Simple Mail Transfer Protocol (SMTP) subscribe connector can be used 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 9.8 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 9.9 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.

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 9.10 Required Parameters for Subscriber MQ Connectors

Parameter	Description
type	Specifies to subscribe.
mqtopic	Specifies the MQ topic name.
mqtype	Specifies binary or CSV.

Table 9.11 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 9.12 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.

Table 9.13 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.

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.

The platform that hosts the running instance of the connector must have the Tervela run-time libraries installed. 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 9.14 Required Parameters for Subscriber Tervela Connectors

Parameter	Description
type	Specifies to subscribe.
tvauserid	Specifies a user name defined in the Tervela TPM. The TPM must have publish-topic entitlement rights associated with this user name.
tvapassword	Specifies the password associated with tvauserid.
tvaprimarystmx	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
<code>tvamaxoutstand</code>	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.
<code>numbufferedmsgs</code>	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.
<code>hotfailover</code>	Enables hot failover mode
<code>urlhostport</code>	Specifies the “ <i>host/port</i> ” string sent in the metadata message published by the connector on topic SAS.META. <i>tvaclientname</i> when it starts.

Table 9.15 Required Parameters for Publisher Tervela Connectors

Parameter	Description
<code>type</code>	Specifies to publish.
<code>tvauserid</code>	Specifies a user name defined in the Tervela TPM. The TPM must have subscribe-topic entitlement rights associated with this user name.
<code>tvapassword</code>	Specifies the password associated with <code>tvauserid</code> .
<code>tvaprimarystmx</code>	Specifies the host name or IP address of the primary TMX.
<code>tvatopic</code>	Specifies the topic name for the topic to which to publish. This topic must be configured on the TPM for the GD service.
<code>tvaclientname</code>	Specifies the client name associated with the Tervela Guaranteed Delivery context. Must be unique among all instances of Tervela connectors.

Parameter	Description
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 9.16 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 will be 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 will be 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.

Table 9.17 *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)

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
```

, which injects them into source windows

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:

- It works with binary event blocks. No other event block formats are supported.

- 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 queue name must be equal to the topic name with “/*buspersistencequeue*” appended, where *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.

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.

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 9.18 Required Parameters for Subscriber Solace Connectors

Parameter	Description
<code>type</code>	Specifies to subscribe.

Parameter	Description
<code>soluserid</code>	Specifies the user name required to authenticate the connector's session with the appliance.
<code>solpassword</code>	Specifies the password associated with <code>soluserid</code> .
<code>solhostport</code>	Specifies the appliance to connect to, in the form " <i>host:port</i> ".
<code>solvpn</code>	Specifies the appliance message VPN to assign the client to which the session connects.
<code>soltopic</code>	Specifies the Solace destination topic to which to publish.
<code>urlhostport</code>	Specifies the <i>host:port</i> field in the metadata topic subscribed to on start-up to field metadata requests.
<code>numbufferedmsgs</code>	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.
<code>hotfailover</code>	Enables hot failover mode.

Table 9.19 Required Parameters for Publisher Solace Connectors

Parameter	Description
<code>type</code>	Specifies to publish.
<code>soluserid</code>	Specifies the user name required to authenticate the connector's session with the appliance.
<code>solpassword</code>	Specifies the password associated with <code>soluserid</code> .
<code>solhostport</code>	Specifies the appliance to connect to, in the form " <i>host:port</i> ".
<code>solvpn</code>	Specifies the appliance message VPN to assign the client to which the session connects.

Parameter	Description
<code>soltopic</code>	Specifies the Solace topic to which to subscribe.
<code>urlhostport</code>	Specifies the <i>host:port</i> field in the metadata topic subscribed to on start-up to field metadata requests.

Table 9.20 *Optional Parameters for Subscriber Solace Connectors*

Parameter	Description
<code>snapshot</code>	Disables the sending of snapshot data. The default is enabled.
<code>collapse</code>	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
<code>buspersistence</code>	Sets the Solace message delivery mode to Guaranteed Messaging. The default value is Direct Messaging.

Table 9.21 *Optional Parameters for Publisher Solace Connectors*

Parameter	Description
<code>buspersistence</code>	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.
<code>buspersistencequeue</code>	Specifies the name appended to the topic name to create the full endpoint topic name.

Tibco Rendezvous (RV) Connector

A 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 platform that hosts the running instance of the connector must have the Tibco RV run-time libraries installed. 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 9.22 Required Parameters for Subscriber Tibco RV Connectors

Parameter	Description
<code>type</code>	Specifies to subscribe.
<code>tibrvsubject</code>	Specifies the Tibco RV subject name.
<code>tibrvtype</code>	Specifies binary or CSV.

Table 9.23 Required Parameters for Publisher Tibco RV Connectors

Parameter	Description
type	Specifies to publish.
tibrvsubject	Specifies the Tibco RV subject name.

Table 9.24 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.

Table 9.25 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.
transactional	Sets the event block type to transactional. The default value is normal.

Parameter	Description
<code>tibrvservice</code>	Specifies the Rendezvous service used by the Tibco RV transport created by the connector. The default service name is “rendezvous”.
<code>tibrvnetwork</code>	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.
<code>tibrvdaemon</code>	Specifies the Rendezvous daemon used by the connector. The default is the default socket created by the local daemon.

User-Written Connectors

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
<code>dfESPconnectorInfo</code>	Specifies the connector name, publish/subscribe type, initialization function pointer, and configuration data pointers.
<code>subRequiredConfig</code>	Specifies an array of <code>dfESPconnectorParmInfo_t</code> entries listing required configuration parameters for a subscriber.
<code>sizeofSubRequiredConfig</code>	Specifies the number of entries in <code>subRequiredConfig</code> .
<code>pubRequiredConfig</code>	Specifies an array of <code>dfESPconnectorParmInfo_t</code> entries listing required configuration parameters for a publisher.
<code>sizeofPubRequiredConfig</code>	Specifies the number of entries in <code>pubRequiredConfig</code> .
<code>subOptionalConfig</code>	Specifies an array of <code>dfESPconnectorParmInfo_t</code> entries listing optional configuration parameters for a subscriber.
<code>sizeofSubOptionalConfig</code>	Specifies the number of entries in <code>subOptionalConfig</code> .
<code>pubOptionalConfig</code>	Specifies an array of <code>dfESPconnectorParmInfo_t</code> entries listing optional configuration parameters for a publisher.
<code>sizeofPubOptionalConfig</code>	Specifies the number of entries in <code>pubOptionalConfig</code> .

A connector must define these static methods:

Static Method	Description
<code>dfESPconnector *initialize(dfESPEngine *engine, dfESPpsLib_t psLib)</code>	Returns an instance of the connector.
<code>dfESPconnectorInfo *getConnectorInfo()</code>	Returns the <code>dfESPconnectorInfo</code> 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
<code>start()</code>	Starts the connector. Must call base class method <code>checkConfig()</code> to validate connector configuration before starting. Must also call base class method <code>start()</code> . Must set variable <code>_started = true</code> upon success.
<code>stop()</code>	Stops the connector. Must call base class method <code>stop()</code> . Must leave the connector in a state whereby <code>start()</code> can be subsequently called to restart the connector.
<code>getState()</code>	Returns the current state of the connector.
<code>callbackFunction()</code>	Specifies the method invoked by the engine to pass event blocks generated by the window to which it is connected.
<code>errorCallbackFunction()</code>	Specifies the method invoked by the engine to report errors detected by the engine. Must call user callback function <code>errorCallback</code> , 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 User-Written 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. User-written 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.

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
```

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

Subscriber use:

```
dfesp_fs_adapter -k sub -h url -t binary | csv | xml | json
<-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 logconfigfile>
```

Publisher use:

```
dfesp_fs_adapter -k pub -h url -f fsname
-t binary | csv | xml | json | syslog <-b blocksize> <-d dateformat>
<-r rate> <-m > <-e> <-i> <-p> <-n> <-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 <i>url</i>	Specify the dfESP publish and subscribe standard URL in the form <code>dfESP://host:port/project/continuousquery/window</code> Append the following for subscribers: <code>?snapshot=true false</code> . Append the following for subscribers if needed: <code>?collapse=true false</code>
—f <i>fsname</i>	Specify the subscriber output file, publisher input file, or socket “ <i>host:port</i> ”. Leave <i>host</i> blank to implement a server.
-t <i>binary csv xml json syslog</i>	Specify the file system type. The syslog value is valid only for publishers.
—c <i>period</i>	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 <i>maxfilesize</i>	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 <i>dateformat</i>	Specify the date format. The default value = %Y-%m-%d %H:%M:%S
-r <i>rate</i>	Specify the requested transmit rate in events per second.
—a <i>aggrsize</i>	Specify, in latency mode, statistics for aggregation block size.
—n	Specify latency mode.
-g <i>gdconfig</i>	Specify the guaranteed delivery configuration file.
—l <i>native solace tervela</i>	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++ <code>C_dfESPPubsubSetPubsubLib()</code> API call.

Parameter	Definition
<code>—j trace debug info warn error fatal off</code>	Set the logging level for the adapter. This is the same range of logging levels that you can set in the <code>C_dfESPpubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
<code>-b blocksize</code>	Set the block size. The default value = 1.
<code>—m maxevents</code>	Specify the maximum number of events to publish.
<code>-e</code>	Specify that events are transactional.
<code>-i</code>	Read from growing file.
<code>-p</code>	Buffer all event blocks before publishing them.
<code>—y logconfigfile</code>	Specify the log configuration file.

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.

Database Adapter

The Database Adapter supports publish and subscribe operations on databases using DataDirect drivers. The adapter is certified for Oracle, DB2, and MySQL platforms. 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.

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".
--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.

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.

SMTP Subscriber Adapter

The SMTP Subscriber Adapter subscribes to only SAS Event Stream Processing Engine windows. Source window publish 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 publish and subscribe standard URL in the form "dfESP://host:port/project/contquery/window. You can append ?snapshot=true false and ?collapse=true false for subscribers
-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++ <code>C_dfESPpubsubSetPubsubLib()</code> 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 <code>C_dfESPpubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
-y logconfigfile	Specify the log configuration file.

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 can occur within a single event stream processor. More likely it occurs across two event stream processors, possibly on different machines on the network. Currently, 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 <i>url</i>	Specify the subscribe standard URL in the form "dfESP:// <i>host:port/project/contquery/window?</i> <i>snapshot=true false>?collapse=true false</i> "
-p <i>url</i>	Specify the publish standard URL in the form dfESP:// <i>host:port/project/contquery/window</i>

The `eventblock` size and transactional nature is inherited from the subscribed window.

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.

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 clients 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 have connectivity 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 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 url\jmstopic
BytesMessage | TextMessage | MapMessage <dateformat> <gdconfigfile >
<native | solace | tervela>< severe | warning | info>
```

Parameter	Definition
— u <i>url</i>	Specify the publish and subscribe standard URL in the form <code>dfESP://host:port/project/contquery/window</code>
—j <i>jmstopic</i>	Specify the JMS topic name.
— d <i>dateformat</i>	Specify the format of <code>ESP_datetime_t</code> and <code>ESP_timestamp_t</code> fields in events. The default is "yyyy-MM-dd HH:mm:ss.SSS".
—l native solace tervela	Specify the transport type. If you specify <code>solace</code> or <code>tervela</code> transports instead of the default <code>native</code> transport, use the required client configuration files specified in the description of the C++ <code>C_dfESPpubsubSetPubsubLib()</code> API call.
- g <i>gdconfigfile</i>	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 <i>url</i>	Specify the publish and subscribe standard URL in the form "dfESP://host:port/project/contquery/window"
—j <i>jmstopic</i>	Specify the JMS topic name.
—b <i>blocksize</i>	Specify the number of events per event block.
-t	Specify that event blocks are transactional. The default is normal.
— d <i>dateformat</i>	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 <i>gdconfigfile</i>	Specify the guaranteed delivery configuration file for the client.
—l <i>native</i> <i>solace</i> <i>tervela</i>	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 <i>severe</i> <i>warning</i> <i>info</i>	Specify the application logging level.

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>
```

Table 9.26 Parameter Definitions

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
-h <i>url</i>	Specify the publish and subscribe standard URL in the form “dfESP://host:port/project/contquery/window” . Append ?snapshot=true false and ?collapse=true false for subscribers if needed.
-f <i>mqtopic</i>	Specify the MQ topic name.
-n <i>mqsubname</i>	Specify the MQ subscription name.
-s <i>mqsubqueue</i>	Specify the MQ queue name.
-q <i>mqqueuemanager</i>	Specify the MQ queue manager name.
-b <i>blocksize</i>	Specify the blocksize. The default value = 1.
-d <i>dateformat</i>	Specify the date format. The default value = %Y-%m-%d %H:%M:%S

Parameter	Definition
<code>-e</code>	Specify that events are transactional.
<code>—g <i>gdconfig</i></code>	Specify the guaranteed delivery configuration file.
<code>—l native solace tervela</code>	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++ <code>C_dfESPPubsubSetPubsubLib()</code> API call.
<code>—j trace debug info warn error fatal off</code>	Set the logging level for the adapter. This is the same range of logging levels that you can set in the <code>C_dfESPPubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
<code>—y <i>logconfigfile</i></code>	Specify the log configuration file.

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 tvaprimarymtx —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 tvaprimarvmtx —f tvatopic
—c tvaclientname -n tvasubname
-o urlhostport <-s tvasecondarvmtx> <-l tvalogfile>
<-g gdconfig> <-l native | solace | tervela>
<-j trace | debug | info | warn | error | fatal | off>
<-y logconfigfile>
```

Table 9.27 Parameter Definitions

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
-h <i>url</i>	Specify the publish and subscribe standard URL in the form “dfESP://host:port/project/contquery/window”. Append ?snapshot=true false and ?collapse=true false for subscribers if needed.
-u <i>tvauserid</i>	Specify the Tervela user name.
-p <i>tvapassword</i>	Specify the Tervela password.
-t <i>tvaprimarvmtx</i>	Specify the Tervela primary TMX.
-f <i>tvatopic</i>	Specify the Tervela topic.
-c <i>tvaclientname</i>	Specify the Tervela client name.
-m <i>tvamaxoutstand</i>	Specify the Tervela maximum number of unacknowledged messages.
-b <i>numbufferedmsgs</i>	Specify the maximum number of messages buffered by a standby subscriber connector.
-o <i>urlhostport</i>	Specify the <i>host:port</i> string sent in connector metadata message
—s <i>tvasecondarvmtx</i>	Specify the Tervela secondary TMX.
—l <i>tvalogfile</i>	Specify the Tervela log file. The default is syslog.

Parameter	Definition
<code>—w tvapubbwlimit</code>	Specify the Tervela maximum bandwidth of published data (Mbps). The default value is 100.
<code>-r tvapubrate</code>	Specify the Tervela publish rate (Kbps). The default value is 30.
<code>—e tvapubmsgexp</code>	Tervela maximum time to cache published messages (seconds); the default value is 1
<code>—g gdconfig</code>	Specify the guaranteed delivery configuration file.
<code>—l native solace tervela</code>	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++ <code>C_dfESPPubsubSetPubsubLib()</code> API call.
<code>—j trace debug info warn error fatal off</code>	Set the logging level for the adapter. This is the same range of logging levels that you can set in the <code>C_dfESPPubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
<code>—y logconfigfile</code>	Specify the log configuration file.

Solace Systems Adapter

The Solace 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>
```

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>
```

Table 9.28 Parameter Definitions

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
-h url	Specify the publish and subscribe standard URL in the form "dfESP://host:port/project/contquery/window". Append "?snapshot= true false" and "?collapse= true false" for subscribers if needed.
-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.

Parameter	Definition
<code>—j trace debug info warn error fatal off</code>	Set the logging level for the adapter. This is the same range of logging levels that you can set in the <code>C_dfESPpubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
<code>—y logconfigfile</code>	Specify the log configuration file.
<code>-b</code>	Use Solace Guaranteed Messaging. By default, Solace Direct Messaging is used.
<code>—q buspersistencequeue</code>	Specify the queue name used by Solace Guaranteed Messaging publisher.

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 9.29 Parameter Definitions

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use.
-h <i>url</i>	Specify the publish and subscribe standard URL in the form <code>"dfESP://host:port/project/contquery/window"</code> . Append <code>"?snapshot= true false "</code> and <code>"?collapse= true false"</code> for subscribers if needed.
-f <i>tibrvsubject</i>	Specify the Tibco RV subject.
-t <i>tibrvtype</i>	Specify “binary” or “CSV”.
-s <i>tibrvservice</i>	Specify the Tibco RV service.
-n <i>tibrvnetwork</i>	Specify the Tibco RV network.
-m <i>tibrvdaemon</i>	Specify the Tibco RV daemon.
-b <i>blocksize</i>	Specify the block size. The default value is 1.
—d <i>dateformat</i>	Specify the date format. The default is %Y-%m-%d %H:%M:%S.
—e	Specify that events are transactional.
—g <i>gdconfig</i>	Specify the guaranteed delivery configuration file.
—l <i>native solace tervela</i>	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++ <code>C_dfESPpubsubSetPubsubLib()</code> API call.
—j <i>trace debug info warn error fatal off</i>	Set the logging level for the adapter. This is the same range of logging levels that you can set in the <code>C_dfESPpubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
—y <i>logconfigfile</i>	Specify the log configuration file.

PI Publisher Adapter

The PI Adapter supports publish operations from a PI server, where it injects a PI event stream identified by a PI tag into a source window. To use the adapter, you must install the PI run-time libraries.

The target source window must have the following schema:

```
"ID*:int32,timestamp:date,value:PItype"
```

Here, PItype is an ESP type that matches the expected PI value type for the configured tag.

Valid mappings are as follows:

Event Stream Processor Data Type	PI Adapter Data Type
ESP_DOUBLE	Float16, Float32, Float64
ESP_INT32	Int16, Int32
ESP_UTF8STR	String
ESP_DATETIME	Timestamp
Not supported	Digital, Blob

Publisher usage:

```
dfesp_pi_adapter -h url -u piuserid -s pihost -t pitag
< -b blocksize><-e> <-g gdconfig>
<-l native | solace | tervela> <-j trace | debug | info | warn | error | fatal>
<-y logconfigfile>
```

Parameter	Definition
-h <i>url</i>	Specify the publish/subscribe standard URL in the form <code>dfESP://host:port/project/contquery/window</code> .

Parameter	Definition
<code>-u piuserid</code>	Specify the PI user name.
<code>—spihost</code>	Specify the PI host.
<code>—t pitag</code>	Specify the PI tag.
<code>-b blocksize</code>	Specify the block size. The default value is 1.
<code>—e</code>	Specify that events are transactional.
<code>—g gdconfig</code>	Specify the guaranteed delivery configuration file.
<code>—l native solace tervela</code>	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++ <code>C_dfESPPubsubSetPubsubLib()</code> API call.
<code>-j trace debug info warn error fatal off</code>	Set the logging level for the adapter. This is the same range of logging levels that you can set in the <code>C_dfESPPubsubInit()</code> publish/subscribe API call and in the engine <code>initialize()</code> call.
<code>—y logconfigfile</code>	Specify the log configuration file.

10

Enabling Guaranteed Delivery

<i>Overview to Guaranteed Delivery</i>	175
<i>Guaranteed Delivery Success Scenario</i>	178
<i>Guaranteed Delivery Failure Scenarios</i>	179
<i>Additions to the Publish/Subscribe API for Guaranteed Delivery</i>	180
<i>Configuration File Contents</i>	180
<i>Publish/Subscribe API Implementation of Guaranteed Delivery</i>	181

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.

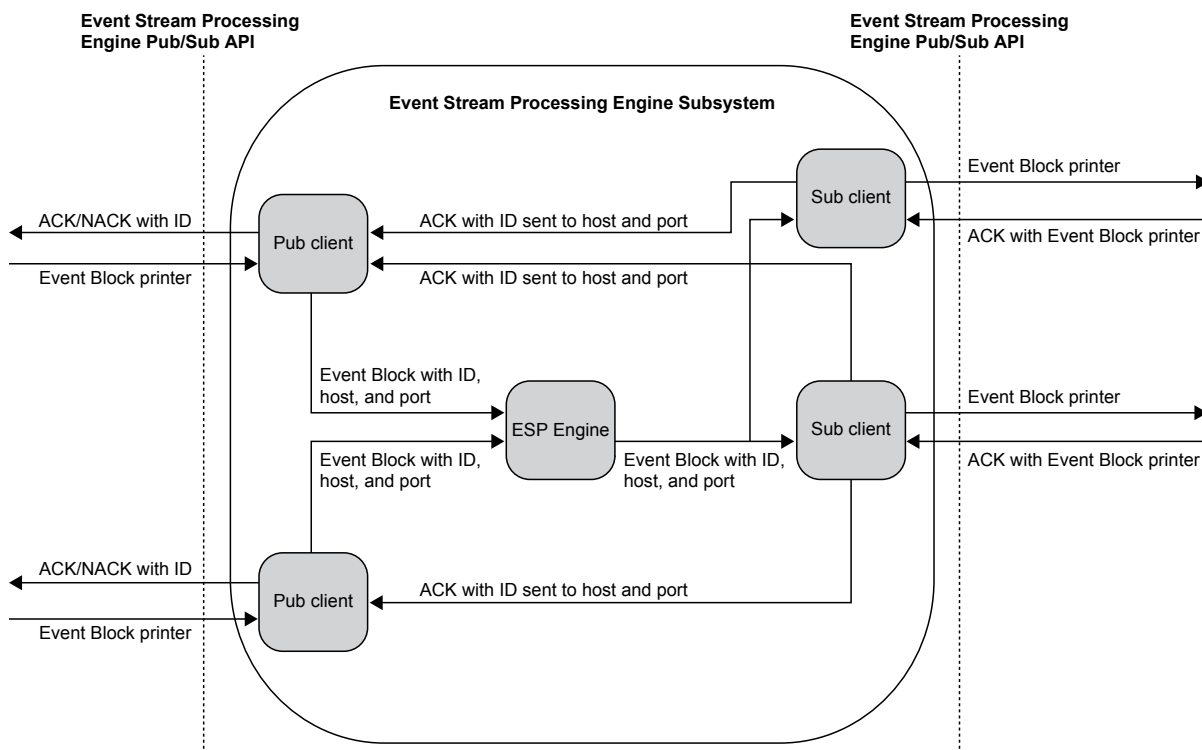
When a publish or subscribe connection is started, a client is established to perform various publish/subscribe activities. When a publish connection is started, the number of guaranteed subscribers required to acknowledge delivery of its event blocks is specified. The time-out value used to generate 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 10.1 *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 all 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.
- It is expected under certain conditions that subscribers will 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 acknowledged/not acknowledged 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	<ul style="list-style-type: none"> ■ 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 <code>NACK</code> and <code>ID</code>. 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	<ul style="list-style-type: none"> ■ 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	<ul style="list-style-type: none"> ■ 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 Publish/Subscribe API” on page 105](#). For publish/subscribe operations that do not have 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.

Guaranteed Delivery-enabled Publisher Configuration File Contents

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

```


11

Implementing 1+N-Way Failover

<i>Overview to 1+N-Way Failover</i>	186
<i>Topic Naming</i>	190
Overview to Topic Naming	190
Solace	190
Tervela	191
<i>Failover Mechanisms</i>	191
Overview to Failover Mechanisms	191
Determining ESP Active/Standby State (Solace)	191
Determining ESP Active/Standby State (Tervela)	193
New ESP Active Actions on Failover (Solace)	194
<i>Restoring Failed Active ESP State after Restart</i>	195
<i>Using ESP Persist/Restore</i>	196
<i>Metadata Exchanges (Solace)</i>	197
<i>Metadata Exchanges (Tervela)</i>	197
<i>Required Software Components</i>	198
<i>Required Client Configuration</i>	198
<i>Required Appliance Configuration (Solace)</i>	199
<i>Required Appliance Configuration (Tervela)</i>	200

Overview to 1+N-Way Failover

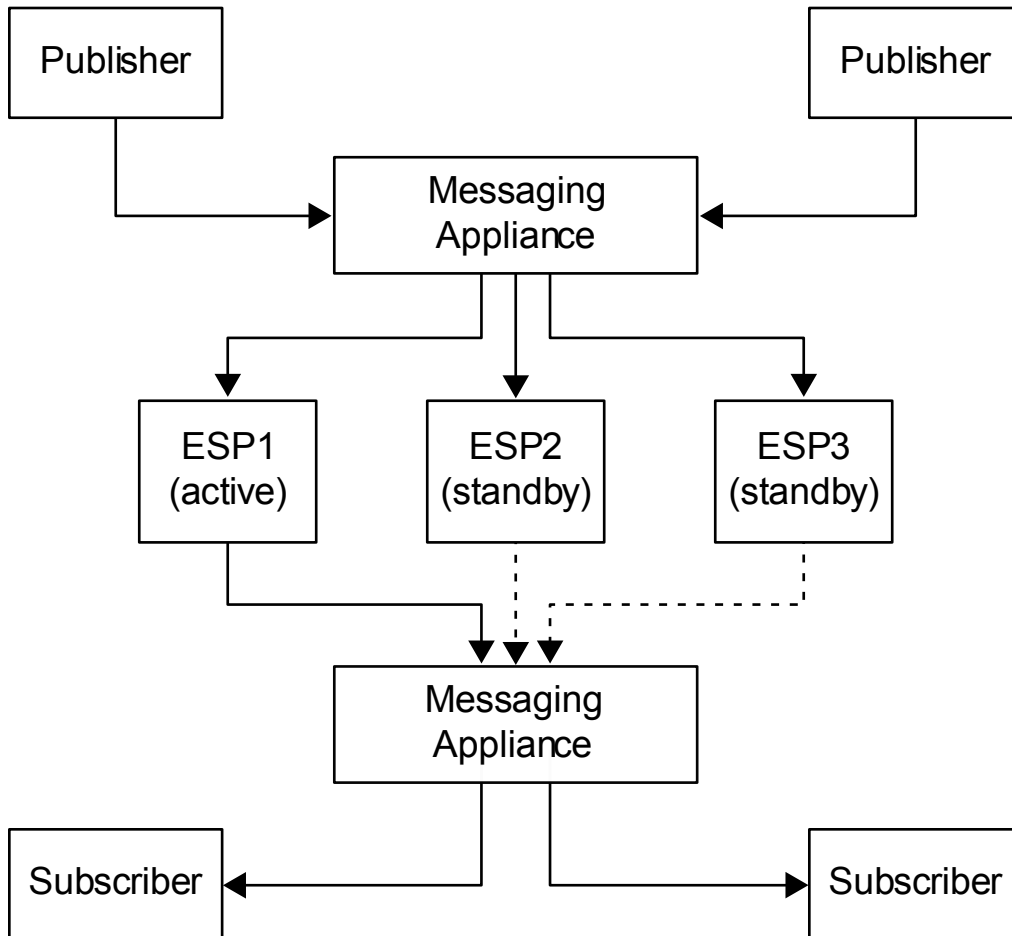
SAS Event Stream Processing Engine can use third-party messaging appliances to provide 1+N-Way Failover. Event stream processing publishers and subscribers work with packages of events called event blocks when they interface with the engine. When traversing a messaging appliance, event blocks are mapped 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 terms messaging appliance, appliance, message fabric, and message bus interchangeably. 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 11.1 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.

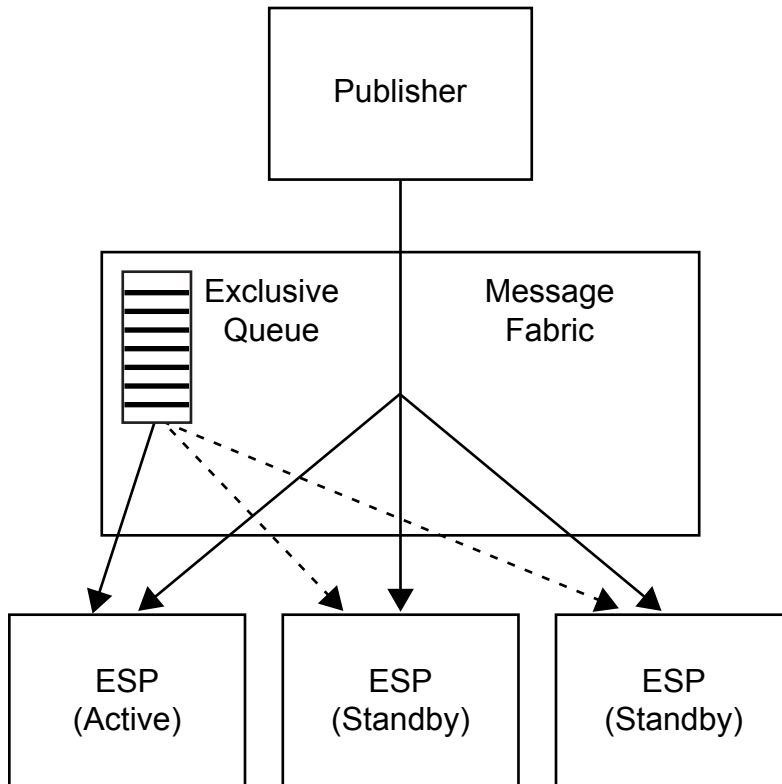
When appliance connectors are inactive, they buffer outbound messages (up to a configurable maximum) so that they can find messages starting with ID+1 in the buffer if necessary.

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 11.2 *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)

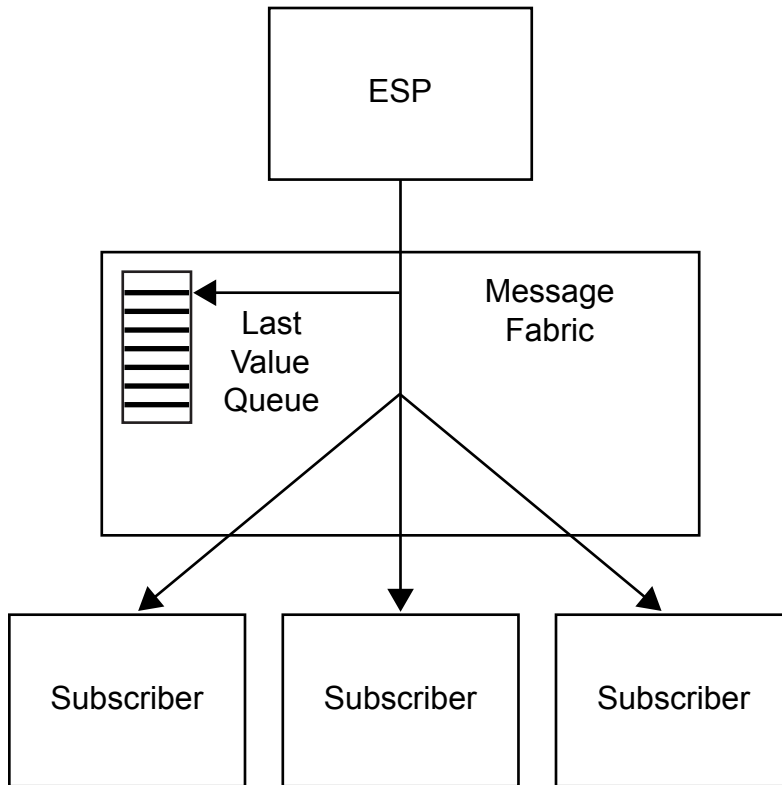
The newly active engine (ESP) determines, from the message bus, the last message published by the previously active ESP for the relevant window. To assist in this process, guaranteed messaging Last Value Queues (LVQs) are used.

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 11.3 *Last Value Queues*



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.

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.

12

Advanced Topics

<i>Logging Bad Events</i>	202
<i>Measuring Time Granularity</i>	202
<i>Using Joins and Understanding Join Constraints</i>	203
<i>Converting CSV Events to Binary</i>	204
<i>Implementing Periodic (or Pulsed) Window Output</i>	205
<i>Splitting Generated Events across Output Slots</i>	206
Overview	206
Splitter Functions	207
Splitter Expressions	208
<i>Marking Events as Partial-Update on Publish</i>	208
Overview	208
Publishing Partial Events into a Source Window	209
Examples	210
<i>Understanding Primary Indexes and Retention Policies</i>	211
Overview	211
Fully Stateful Indexes	211
Retention Policies for Fully Stateful Indexes	213
Non-Stateful Index	213
<i>Using Stateless Pattern Windows</i>	214
<i>Using Aggregation Functions</i>	215

<i>Using an Aggregate Function to Add Statistics to an Incoming Event</i>	218
<i>Persist and Restore Model State</i>	220
<i>Gathering and Saving Latency Measurements</i>	221

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.

Using Joins and 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.	<ul style="list-style-type: none"> ■ 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).
M-1	Fact	Dimension	All keys on right and not all keys on left participate in the constraint set.	<ul style="list-style-type: none"> ■ 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.	<ul style="list-style-type: none"> ■ 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

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 should be used for a given event. This enables you to split generated events across a set of output slots. This determines where the subsequent set of windows goes.

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. Using splitter functions can be more efficient than using filter windows off the output slot, especially if you are doing an alpha-split across a set of trades or similar task.

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.

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. You apply the partial event Update in the source window.

When you use transactional event blocks to publish multiple events atomically, it is invalid to include the Insert of the original event with unique key fields. It is also invalid to include an Update to that same event (identified by the key fields) in the same publishing block. This is 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
<code>pi_RBTREE</code>	Specifies red/black tree, logarithmic Insert, Deletes are $O(\log(n))$ — provides smooth latencies.

Index Type	Description
<code>pi_HASH</code>	Specifies a typical open hash algorithm. In general this index provides faster results than <code>pi_RBTREE</code> . Unless properly sized, using this index might lead to latency spikes.
<code>pi_CL_HASH</code>	Specifies a closed hash. This index provides faster results than <code>pi_HASH</code> .
<code>pi_FW_HASH</code>	Specifies a forward hash. This index creates a smaller memory footprint than other hash indexes, but might yield poorer delete performance.
<code>pi_LN_HASH</code>	Specifies a linked hash. This index performs slightly more slowly than other hash index and uses more memory than <code>pi_CL_HASH</code> .

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()` call on the window. This call takes two parameters, the retention type and the retention parameter.

Retention type can be one of the following options:

Retention Type	Description
<code>ret_NONE</code>	This is the default. It is the same as if <code>setRetentionParms</code> was not called.
<code>ret_BYTIME_SLIDING</code>	Specifies to purge continuously from the window when an event age reaches the limit as specified in the retention parameter.
<code>ret_BYTIME_JUMPING</code>	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.
<code>ret_BYCOUNT_SLIDING</code>	Specifies to purge continuously from the window when the event set size reaches the limit as specified by the retention parameter.
<code>ret_BYCOUNT_JUMPING</code>	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.

Non-Stateful Index

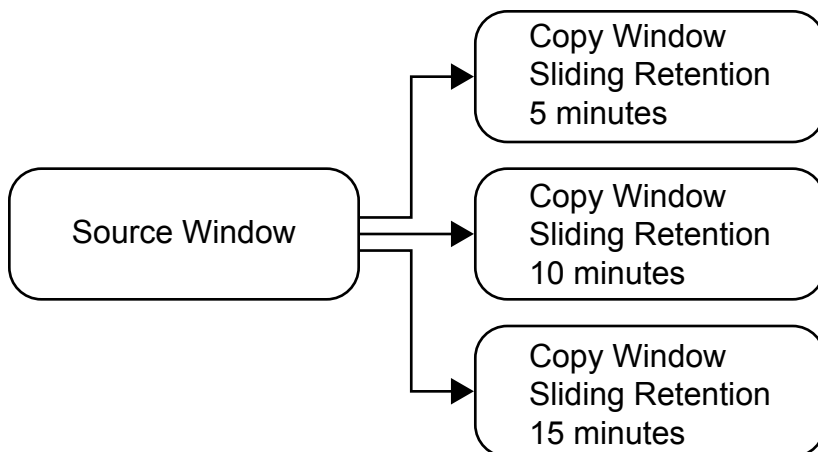
The non-stateful index is a source window that can be set to use the index type `pi_EMPTY`. It acts as a pass-through for all incoming events.

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 22](#).
- 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 12.1 Copy Windows



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, and they reject records that are not Inserts. 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 non-Insert data and allow 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]
```

Using Aggregation Functions

Consider the aggregate function `Sum()`. Even though it is always possible to iterate over the group and compute a new `Sum()` when a new group changes, it is faster to maintain the `Sum()` in a `dfESPdatavar` in the `dfESPgroupstate` object and increment (modify) 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.

Functions that compute themselves based on previous field state and a new field value are called additive aggregation functions. Some functions cannot be made additive, like the function `Max()`. Given a mix of inserts, updates, and deletes, the `Max()` function must iterate over the group when an update or delete is received to correctly compute the aggregate value.

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()` aggregate 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 a 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 a 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. First, they must look at the current state (for example, the last computed state). Second, they must evaluate the type of incoming event to make proper adjustments. 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 aggregations 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.

Using an Aggregate Function to Add Statistics to an Incoming Event

You can use the `ESP_aLast(fieldName)` aggregate function to add aggregate statistics directly to an incoming event by passing the incoming event fields through the aggregate window. Alternatively, you can add aggregate statistics by adding a join window after the aggregate window. Adding a join window in this way joins the aggregate calculations or event to the same event that feeds into the aggregate window. That might not produce optimal results.

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.

Persist and Restore Model State

SAS Event Stream Processing Engine includes a `dfESPpersist` class. An instance of this class enables you to persist the complete model state to a file system. It also enables you to restore a model from a persist directory that had been created by a previous persist operation.

To create a persist object, 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. Then 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.

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
<code>void set_playbackRate(int32_t r)</code>	Sets the requested inject rate.
<code>void set_bucketSize(int32_t bs)</code>	Sets the <code>bucketSize</code> parameter previously described.
<code>void set_maxEvents(int32_t me)</code>	Sets the maximum number of events to inject.
<code>void set_oFile(char *ofile)</code>	Sets the name of the output file containing latency statistics.

Method	Description
<code>void set_iFile(char *ifile)</code>	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
<code>void set_injectPoint(dfESPwindow_source *s)</code>	Sets the source window in which you want events time stamped with a beginning timestamp.
<code>void read_and_buffer()</code>	Reads the input event blocks from the configured input file and buffers them.
<code>void playback_at_rate()</code>	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	
<code>—r rate</code>	Specifies the requested transmit rate in events per second.
<code>-m maxevents</code>	Specifies the maximum number of events to publish.
<code>-p</code>	Specifies to buffer all events prior to publishing.
<code>-n</code>	Enables latency mode.

Subscriber	
<code>-r rate</code>	Specifies the requested transmit rate in events per second.
<code>-a aggrsize</code>	Specifies the aggregation bucket size.
<code>-n</code>	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.

Appendix 1

Example: Implementation of the Trades Model

The following C++ program implements a model containing the continuous query described in [“Understanding Continuous Queries” on page 5](#). The program follows these steps.

- 1 Include needed header files from the SAS Event Stream Processing Engine library so that you can use the appropriate objects to define elements of the model.
- 2 After the main declaration, define objects for the engine, the project, the event depot, and the continuous query.
- 3 Set the date format for the engine.
- 4 Define the Trades and Traders source windows.
- 5 Define the LargeTrades filter window.
- 6 Define the AddTraderName join window. Specify the join conditions.
- 7 Define the TotalCost compute window.
- 8 Define the AddSecurity aggregate window.
- 9 Set the number of threads for the engine. Run the project. Shut down the engine.

```
#include "dfESPengine.h"  
#include "dfESEventdepot_mem.h"  
#include "dfESPwindow_source.h"
```

```

#include "dfESPwindow_aggregate.h"
#include "dfESPwindow_filter.h"
#include "dfESPwindow_join.h"
#include "dfESPwindow_compute.h"
#include "dfESPcontquery.h"
#include "dfESPproject.h"

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

    dfESPengine      *theEngine
    = dfESPengine::initialize(argc, argv, "theEngine", pubsub_DISABLE);
    dfESPproject      *project_01
    = theEngine->newProject("project_01");
    dfESPeventdepot_mem *mem_01
    = project_01->newEventdepot_mem("mem_01");
    dfESPcontquery     *cq_01
    = project_01->newContquery("cq_01");

    theEngine->set_dateFormat((char *) "%d/%b/%Y:%H:%M:%S");

    dfESPwindow_source *Trades
    = cq_01->newWindow_source("Trades", mem_01, dfESPindextypes::pi_HASH,
        dfESPstring("tradeID*:string,security:string,quantity:int32,price:double,
            traderID:int64,time:stamp"));

    dfESPwindow_source *Traders
    = cq_01->newWindow_source("Traders", mem_01,
        dfESPindextypes::pi_HASH,
        dfESPstring("ID*:int64,name:string"));

    dfESPwindow_filter *LargeTrades
    = cq_01->newWindow_filter("LargeTrades", mem_01,
        dfESPindextypes::pi_RBTREE);
    LargeTrades->setFilter("quantity>=100");

    dfESPwindow_join *AddTraderName
    = cq_01->newWindow_join("AddTraderName", dfESPwindow_join::jt_LEFTOUTER,
        mem_01, dfESPindextypes::pi_RBTREE);
    AddTraderName->setJoinConditions("l_traderID=r_ID");
    AddTraderName->setJoinSelections("l_security,l_quantity,l_price,l_traderID,l_time,r_name");
    AddTraderName->setFieldSignatures("security:string,quantity:int32,price:double,traderID:int64,
        time:stamp,name:string");

    dfESPwindow_compute *TotalCost
    = cq_01->newWindow_compute("TotalCost", mem_01,
        dfESPindextypes::pi_RBTREE,
        dfESPstring("tradeID*:string,security:string,quantity:int32,price:double,totalCost:double,

```

```

        traderID:int64,time:stamp,name:string"));
TotalCost->addNonKeyFieldCalc("security");
TotalCost->addNonKeyFieldCalc("quantity");
TotalCost->addNonKeyFieldCalc("price");
TotalCost->addNonKeyFieldCalc("price*quantity");
TotalCost->addNonKeyFieldCalc("traderID");
TotalCost->addNonKeyFieldCalc("time");
TotalCost->addNonKeyFieldCalc("name");

dfESPwindow_aggregate    *BySecurity
= cq_01->newWindow_aggregate("BySecurity", mem_01, dfESPindextypes::pi_RBTREE,
                             dfESPstring("security*:string,quantityTotal:double,
                                           costTotal:double"));
BySecurity->addNonKeyFieldCalc("ESP_aSum(quantity)");
BySecurity->addNonKeyFieldCalc("ESP_aSum(totalCost)");

cq_01->addEdge(Trades, 0, LargeTrades);
cq_01->addEdge(LargeTrades, 0, AddTraderName);
cq_01->addEdge(Traders, 0, AddTraderName);
cq_01->addEdge(AddTraderName, 0, TotalCost);
cq_01->addEdge(TotalCost, 0, BySecurity);

project_01->setNumThreads(1);
theEngine->startProjects();

dfESPengine::shutdown();

return 0;
}

```


Appendix 2

Example: Subscriber and Publisher Applications

Subscriber Application

This example can also be found in the release installation under `$DFESP_HOME/src`.

```
/* Standard C include statements */
#include <stdlib.h>
#include <string.h>
/* The following include file is needed for all subscribing code. */
#include "C_dfESPpubsubApi.h"

/* The following include files are specific to the ESP objects being used here. */
#include "C_dfESPevent.h"
#include "C_dfESPstringV.h"

/* Flag indicating whether to keep main in a non-busy wait
   while the subscribeFunction runs.
*/
typedef struct {
    unsigned short nonBusyWait;
} callback_ctx;

/* Define a subscribe function that gets called when new events
are published from the server via the pub/sub API. This function gets an
eventblock and the schema of the event for processing.
For this example, write the event as CSV to stdout.
*/
void subscribeFunction(C_dfESPeventblock eb, C_dfESPschema schema, void *ctx) {
    C_dfESPevent ev;
    C_ESP_int32_t eventCnt = C_dfESPeventblock_getSize(eb);
    C_ESP_int32_t eventIdx;
```

```

C_ESP_utf8str_t eventCSV;
callback_ctx *context = (callback_ctx *)ctx;

for (eventIndx=0; eventIndx < eventCnt; eventIndx++) {
    /* Get the event out of the event block. */
    ev = C_dfESPeventblock_getEvent(eb, eventIndx);

    /* Convert from binary to CSV using the schema & print to stdout. */
    eventCSV = C_dfESPevent_toStringCSV_noBuff(ev, schema, 1);
    if (!eventCSV) {
        fputs("C_dfESPevent_toStringCSV() failed\n", stderr);
        C_dfESPeventblock_destroy(eb);
        context->nonBusyWait = 0; /* Release the busy wait which ends the program. */
        return;
    }

    printf("%s\n", eventCSV);

    free(eventCSV);

    if (C_dfESPevent_getOpcode(ev) == eo_UPDATEBLOCK)
        ++eventIndx; /* skip the old record in the update block */
}

/* Flush stdout */
fflush(stdout);

/* Free the event block and its contained events.
This must be done by the subscribe callback function, else major memory leak.
*/
C_dfESPeventblock_destroy(eb);
}

/* Optionally define a callback function for subscription failures given
you might want to try to reconnect/recover. Here, just print
some error information and release the non-busy wait set below so the
main in program can end.
*/
void subFailureFunction(C_dfESPpubsubFailures failure,
                      C_dfESPpubsubFailureCodes code, void *ctx)
{
    callback_ctx *context = (callback_ctx *)ctx;

    /* Don't output anything for normal server disconnect */
    if (failure != pubsubFail_SERVERDISCONNECT) {
        char *failMsg = C_dfESPdecodePubSubFailure(failure);
        char *codeMsg = C_dfESPdecodePubSubFailureCode(code);
    }
}

```



```

    fputs("Client subscription error: ", stderr);
    fputs(failMsg, stderr);
    fputs(codeMsg, stderr);
    fputs("\n", stderr);
}

/* Release the busy wait, which ends the program. */
context->nonBusyWait = 0;
}

/* This is a subscribe client example for using the ESP pub/sub API. You could test this
against the subscribeServer example provided in the ESP server distributions
This is a generic subscribe tool to subscribe to any window's events for any
instance of an ESP application, which could also be a server.
*/
int main(int argc, char* argv[]) {
    short schemaOnly = 0;
    int len;
    char *url;
    int i;
    C_dfESPstringV schemaVector;
    clientObjPtr sub;
    callback_ctx cbContext;
    cbContext.nonBusyWait = 1;

    /* Check command line arguments. */
    if ((argc < 2) || (argc > 3)) {
        fputs("\nUsage: subscribe_client <url>, where schema is for schema only and <url>
is of the form:
dfESP://<host>:<port>/<project>/<continuousquery>/<window>?snapshot=<true/false>
?[collapse=true]\n\n", stderr);
return -1;
    }

    /* Determine whether you want the schema & events or just the schema. */
    if (argc == 3) {
        if (strcmp(argv[2], "schema") == 0)
            schemaOnly = 1;
        else {
            fputs("The 2nd argument is invalid, it should be schema\n", stderr);
            return -1;
        }
    }

    /* Initialize subscribing capabilities. This is the first pub/sub API call that must
be made, and it only needs to be called once. The first parameter is the log level,
so turn logging off. The second parameter provides the ability to provide a log

```

config filepath to configure how logging works, NULL uses the defaults, but this does not matter given it's getting turned off.

```

*/
if (C_dfESPpubsubInit(ll_Off, NULL) != 1) {
    fputs("C_dfESPpubsubInit() failed\n", stderr);
    return -1;
}

/* Get the schema and write it to stdout. The URL is as follows:
dfESP://hostname:port/project_name/continuous_query_name/window_name?get=schema
Remove the snapshot extension from the passed url and substitute "?get=schema".
*/
len = strlen(argv[1]);
url = malloc(len + strlen("?get=schema") + 1);
if (url == NULL) {
    fputs("malloc() failed\n", stderr);
    return -1;
}
strcpy(url, argv[1]);
for (i = 0; i < len; i++) {
    if (url[i] == '?')
        break;
}
strcpy( &url[i], "?get=schema");
schemaVector = C_dfESPpubsubQueryMeta(url);
if (schemaVector == NULL) {
    fputs("Schema query (", stderr);
    fputs(url, stderr);
    fputs(") failed\n", stderr);
    free(url);
    return -1;
}
free(url);
printf("%s\n", C_dfESPstringV_get(schemaVector, 0));
fflush(stdout); /* Flush stdout */
C_dfESPstringV_free(schemaVector); /* free up schema vector & elements, else memory leak */
if (schemaOnly)
    return 0; /* Only return the schema, no events. */

/* Start this subscribe session. This validates subscribe connection parameters,
but does not make the actual connection.
The parameters for this call are URL, user defined subscriber callback function, and
an optional user defined subscribe services error callback function.
The URL is dfESP://hostname:port/project_name/continous_query_name/window_name
When a new window event arrives, the callback function is invoked to process it.
*/

```

```

sub = C_dfESPsubscriberStart(argv[1], subscribeFunction, subFailureFunction, &cbContext);
if (sub == NULL) {
    fputs("C_dfESPsubscriberStart(", stderr);
    fputs(argv[1], stderr);
    fputs(", C_dfESPsubscriberCB_func, C_dfESPpubsubErrorCB_func) failed\n", stderr);
    return -1;
}

/* Now make the actual connection to the ESP application or server. */
if (C_dfESPpubsubConnect(sub) != 1) {
    fputs("C_dfESPpubsubConnect() failed\n", stderr);
    C_dfESPpubsubStop(sub, 0);
    return -1;
}

/* Create a mostly non-busy wait loop. */
while (cbContext.nonBusyWait) {
    C_dfESPlibrary_gSleep(1);
}

/* Stop pubsub, but block (i.e., true) to ensure that all queued events are first processed. */
C_dfESPpubsubStop(sub, 1);

return 0;
}

```

Publisher Application

This example can also be found in the release distribution under `$DFESP_HOME/src`.

```

/* This include file is needed for all publishing code. */
#include "C_dfESPpubsubApi.h"

/* These include files are specific to the ESP objects being used here. */
#include "C_dfESPlibrary.h"
#include "C_dfESPstringV.h"
#include "C_dfESPschema.h"

/* std includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Optionally you can define a callback function for publisher failures given

```

you might want to try to reconnect/recover, but in this example just print some error information. The cbf has an optional context pointer for sharing state across calls or passing state into calls.

```

*/
void clientFailureFunction(C_dfESPpubsubFailures failure,
                          C_dfESPpubsubFailureCodes code,void *ctx) {
    char *failMsg;
    char *codeMsg;

    /* Don't output anything for client busy */
    if ((failure == pubsubFail_APIFAIL) &&
        (code == pubsubCode_CLIENTEVENTSQUEUED)) {
        return;
    }
    failMsg = C_dfESPdecodePubSubFailure(failure);
    codeMsg = C_dfESPdecodePubSubFailureCode(code);
    fputs("Client services error: ", stderr);
    fputs(failMsg, stderr);
    fputs(codeMsg, stderr);
    fputs("\n", stderr);
}

/* This is a publisher client example for using the ESP pub/sub API. The events that
are published to the server are read from a file, so this application needs a binary
event file supplied to it. One could test this against the subscribeServer example
provided in the ESP server distributions. If doing so, simply use the same event input
file already provided for subscribeServer, which ensures that the events published by
this client are valid for a window in the subscribeServer example. This also requires
that the continuous query and window command line parameters are "subscribeServer"
and "tradesWindow" respectively. Finally, to isolate events to those published by this
client, simply run the server without providing the event input file.
*/
int main(int argc, char* argv[]) {
    clientObjPtr pub;
    C_dfESPstringV schemaVector;
    C_dfESPeventblock ib;
    C_dfESPschema schema;
    char *url;
    int len;
    int rc;

    /* Check command line arguments. */
    if (argc != 2) {
        fputs("\nUsage: publish_client <url> <file.bin, where <url> is of the form:
        dfESP://<host>:<port>/<project>/<continuousquery>/<window>\n\n", stderr);
        return -1;
    }

```

```

}

/* Initialize publishing capabilities. This is the first pub/sub API call that must
be made, and it only needs to be called once. The first parameter is the log level,
so we are turning logging off for this example. The second parameter provides
the ability to provide a log config filepath to configure how logging works, NULL
will use the defaults, but for us this does not matter given we're turning it off.
*/
if (C_dfESPpubsubInit(ll_Off, NULL) != 1) {
    fputs("C_dfESPpubsubInit() failed\n", stderr);
    return -1;
}

/* Get the window schema string. The URL for this is as follows:
dfESP://hostname:port/project_name/continous_query_name/window_name?get=schema
*/
len = strlen(argv[1]);
url = malloc(len + strlen("?get=schema") + 1);
if (url == NULL) {
    fputs("malloc() failed\n", stderr);
    return -1;
}
strcpy(url, argv[1]);
strcat(url, "?get=schema");
schemaVector = C_dfESPpubsubQueryMeta(url);
if (schemaVector == NULL) {
    fputs("Schema query (", stderr);
    fputs(url, stderr);
    fputs(") failed\n", stderr);
    free(url);
    return -1;
}
free(url);

/* Use the schema string to create the schema object needed to pass
to C_dfESPEventblock_newEventBlock2() below */
schema = C_dfESPschema_create1("window_schema", C_dfESPstringV_get(schemaVector, 0));
if (!schema) {
    fputs("C_dfESPschema_create1() failed for schema ", stderr);
    fputs(C_dfESPstringV_get(schemaVector, 0), stderr);
    fputs("\n", stderr);
    C_dfESPstringV_free(schemaVector);
    return -1;
}

/* Free up the vector, else memory leak */
C_dfESPstringV_free(schemaVector);

```

```

/* Start this publish session. This validates publish connection parameters,
but does not make the actual connection.
The parameter for this call is the following URL which was provided as an argument:
dfESP://hostname:port/project_name/continous_query_name/window_name
*/
pub = C_dfESPpublisherStart(argv[1], clientFailureFunction, NULL);
if (pub == NULL) {
    fputs("C_dfESPpublisherStart(", stderr);
    fputs(url, stderr);
    fputs(", C_dfESPpubsubErrorCB_func) failed\n", stderr);
    C_dfESPschema_delete(schema);
    return -1;
}

/* Now make the actual connection to the ESP application or server. */
rc = C_dfESPpubsubConnect(pub);
if (rc != 1) {
    fputs("C_dfESPpubsubConnect() failed\n", stderr);
    C_dfESPpubsubStop(pub, 0);
    C_dfESPschema_delete(schema);
    return -1;
}

/* ***** NOW INJECT TRADE EVENTS INTO THE RUNNING PROJECT ***** */

/* Trade event blocks are in binary form and are coming via stdin. */

// for windows it is essential that we read binary data in in BINARY mode.
//
C_dfESPlibrary_setBinaryMode(stdin);

while (1) { /* more trades exist */
    /* Create event block. */
    ib = C_dfESPeventblock_newEventBlock2(stdin, schema);

    if (feof(stdin))
        break;

    rc = C_dfESPpublisherInject(pub, ib);
    C_dfESPeventblock_destroy(ib);
    if (rc != 1) {
        fputs("C_dfESPpublisherInject() failed\n", stderr);
        C_dfESPschema_delete(schema);
        C_dfESPpubsubStop(pub, 0);
        return -1;
    }
}

```

```
}

/* Delete the schema object */
C_dfESPschema_delete(schema);

/* Stop pubsub, but block (i.e., true) to ensure that all queued events
are first processed. */
C_dfESPpubsubStop(pub, 1);

return 0;
}
```


Appendix 3

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 Quality Knowledge Base product.

```
// -*- 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
```

242 Appendix 3 / Example: Using Blue Fusion Functions

```
// 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 continuous 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 4

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 A4.1 Logging Level for the Adapter

Logging Level	Parameter Setting	Description	Written To
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
ERROR	dfESPLLError	Report on other run-time errors or unexpected conditions.	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

Logging Level	Parameter Setting	Description	Written To
FATAL	dfESPLLFatal	Report severe errors that cause premature termination.	Immediately visible on the console
OFF	dfESPLLOff	Logging is turned off.	NA

Glossary

complex event processing (CEP)

event processing that assumes an event cloud as input and therefore makes no assumptions about the arrival order of events.

continuous query

a model of source, relational, pattern matching, and procedural windows. The windows represent continuously executing query-generating updates to the query result set as new events stream through.

Delete (D)

an opcode that removes event data from a window. Similar to an SQL DELETE statement, which removes one or more records from a table.

derived windows

windows that display events that have been fed through other windows and that perform computations or transformations on these incoming events. The current derived window types are aggregate, compute, copy, filter, join, pattern matching, procedure, and union.

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. There can be only one of these in a given model. The engine manages the project resources.

event

a packet of data accessible as a collection of fields, with at least one of those fields defined as the key.

event block

a grouping or package of events. It has a unique embedded transaction ID generated as it is absorbed into a continuous query. It also has a unique ID that an event stream publisher can assign to it.

event stream processing

a process that enables real-time decision making by continuously analyzing large volumes of data as it is received. It is a subcategory of complex event stream processing (CEP).

event stream processing application

a SAS-based event stream processing application embeds a SAS Event Stream Processing Engine with its own dedicated thread pool (or pools) into its application space. This is where the main application thread can do application-specific work and the ESP processes the event streams by creating an instance of the model into processing objects.

event streams

a continuous flow of events (or more precisely, event blocks).

factory server

a server for factory objects, which 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.

Insert (I)

an opcode that adds event data to a window. Similar to an SQL INSERT statement, which adds one or more records to any single table.

memory depot

a repository for indexes and event data that is used by a project.

model

a user specification of one or more data flows used to transform input event streams into meaningful output event streams. A model is represented as a directed graph.

modeling API

a SAS C library that enables developers to write event stream processing models.

opcode

See [“operation code”](#).

operation code

an instruction that specifies an action to be performed.

project

the second-level container in a model (managed by the parent engine container) where the thread pool size is set and contains one or more continuous queries. You can have multiple projects in an engine instance.

publish/subscribe API

a library that enables you to publish event streams into an event stream processor or 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.

schema

a model of the overall structure of data to be processed in a window.

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.

Update (U)

an opcode that changes event data in a window. Similar to an SQL UPDATE statement, which changes the data of one or more records in a table.

Upsert (P)

an opcode that updates event data in a window if the key field already exists and otherwise inserts event data. Similar to an SQL MERGE function in which data for a record is updated, inserted, or both.

window

a processing node in an event stream processing model. Source and derived windows form a continuous query.