THE POWER TO KNOW.

SAS® Event Stream Processing 3.1

User's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2015. *SAS® Event Stream Processing 3.1: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Event Stream Processing 3.1: User's Guide

Copyright © 2015, 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 License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a) and DFAR 227.7202-4 and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

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

May 2015

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 trademarks of their respective companies.

Contents

About This Book . . . . . *vii*
What's New in SAS Event Stream Processing . . . . . *ix*

Chapter 1 • Overview to SAS Event Stream Processing **1**
Product Overview 1
Getting Started with SAS Event Stream Processing 2
Writing an Application with SAS Event Stream Processing 5

Chapter 2 • Understanding Event Stream Processing Modeling Objects **7**
Designing an Event Stream Processing Application 7
What Is a Model? 8
Understanding Events 9
Understanding Event Blocks 10
Implementing Engines 11
Understanding Projects 12
Understanding Continuous Queries 12
Understanding Windows 13

Chapter 3 • Using the XML Layer **17**
Using the XML Server 18
Injecting Events Directly into the XML Server 26
Using the XML Client 31
Validating Your XML Code 36
XML Language Elements 36
XML Code Examples 84

Chapter 4 • Using SAS Event Stream Processing Studio **89**
Overview to SAS Event Stream Processing Studio 89
Using SAS Event Stream Processing Studio 89
Example: Creating a Two-Window Model 91

Chapter 5 • Programming with the C++ Modeling API **95**
Overview to the C++ Modeling API 95
Dictionary 96

Chapter 6 • Using Expressions **123**
Overview to Expressions 123
Understanding Data Type Mappings 124
Using Event Metadata in Expressions 124
Using DataFlux Expression Language Global Functions 125
Using Blue Fusion Functions 126

Chapter 7 • Creating Aggregate Windows **127**
Overview to Aggregate Windows 127
Flow of Operations 128
Using Aggregate Functions 128
XML Examples of Aggregate Windows 138

Chapter 8 • Creating Counter Windows 141
Overview to Counter Windows 141
Examples 142

Chapter 9 • Creating Functional Windows 145
Overview to Functional Windows 145
Using Event Loops 145
Understanding and Using Function Context 146
Functional Window Examples 151

Chapter 10 • Creating Join Windows 153
Overview to Join Windows 153
Understanding Streaming Joins 154
Creating Empty Index Joins 157
Examples of Join Windows 157

Chapter 11 • Creating Notification Windows 159
Overview to Notification Windows 159
Notification Window Delivery Channels 161
Using the Function-Context Element 164
Examples of Notification Windows 167

Chapter 12 • Creating Pattern Windows 175
Overview of Pattern Windows 175
State Definitions for Operator Trees 178
Restrictions on Patterns 180
Using Stateless Pattern Windows 182
Pattern Window Examples 182

Chapter 13 • Creating Procedural Windows 191
Overview to Procedural Windows 191
Using C++ Window Handlers 192
Using DS2 Window Handlers 194
XML Examples of Procedural Windows 198

Chapter 14 • Advanced Window Operations 201
Implementing Periodic (or Pulsed) Window Output 201
Splitting Generated Events across Output Slots 202
Marking Events as Partial-Update on Publish 204
Understanding Retention 206
Understanding Primary and Specialized Indexes 208
Persist and Restore Operations 215
Gathering and Saving Latency Measurements 217

Chapter 15 • Using the Publish/Subscribe API 221
Overview to the Publish/Subscribe API 221
Understanding Publish/Subscribe API Versioning 222
Using the C Publish/Subscribe API 223
Using the Java Publish/Subscribe API 237
Publish/Subscribe API Support for Google Protocol Buffers 243
Publish/Subscribe API Support for JSON Messaging 246

Chapter 16 • Using Connectors 249
Overview to Using Connectors 250
Using the Database Connector 254
Using File and Socket Connectors 259

Using the IBM WebSphere MQ Connector . . . 265
Using the PI Connector . . . 267
Using the Project Publish Connector . . . 271
Using the Rabbit MQ Connector . . . 271
Using the SMTP Subscribe Connector . . . 276
Using the Sniffer Connector . . . 277
Using the Solace Systems Connector . . . 280
Using the Teradata Connector . . . 284
Using the Tervela Data Fabric Connector . . . 286
Using the Tibco Rendezvous (RV) Connector . . . 290
Writing and Integrating a Custom Connector . . . 293

Chapter 17 • Using Adapters . . . **297**
Overview to Adapters . . . 297
Using the Database Adapter . . . 298
Using the Event Stream Processor Adapter . . . 300
Using the File and Socket Adapter . . . 300
Using the IBM WebSphere MQ Adapter . . . 303
Using the HDAT Reader Adapter . . . 304
Using the HDFS (Hadoop Distributed File System) Adapter . . . 305
Using the Java Message Service (JMS) Adapter . . . 308
Using the SAS LASR Analytic Server Adapter . . . 311
Using the PI Adapter . . . 314
Using the Rabbit MQ Adapter . . . 315
Using the REST Subscriber Adapter . . . 317
Using the SAS Data Set Adapter . . . 318
Using the SMTP Subscriber Adapter . . . 321
Using the Sniffer Publisher Adapter . . . 322
Using the Solace Systems Adapter . . . 323
Using the Teradata Subscriber Adapter . . . 325
Using the Tervela Data Fabric Adapter . . . 327
Using the Tibco Rendezvous (RV) Adapter . . . 329
Using the Twitter Publisher Adapter . . . 331

Chapter 18 • Visualizing Event Streams . . . **337**
Overview to Event Visualization . . . 337
Using Streamviewer . . . 337

Chapter 19 • Enabling Encryption on Socket Connections . . . **339**
Overview to Enabling Encryption . . . 339
Understanding SSL Certificate Requirements . . . 340
Understanding the SSL Handshake Process . . . 340

Chapter 20 • Enabling Guaranteed Delivery . . . **341**
Overview to Guaranteed Delivery . . . 341
Guaranteed Delivery Success Scenario . . . 343
Guaranteed Delivery Failure Scenarios . . . 344
Additions to the Publish/Subscribe API for Guaranteed Delivery . . . 344
Configuration File Contents . . . 345

Chapter 21 • Implementing 1+N-Way Failover . . . **347**
Overview to 1+N-Way Failover . . . 347
Topic Naming . . . 350
Failover Mechanisms . . . 351
Restoring Failed Active ESP State after Restart . . . 355
Using ESP Persist/Restore . . . 355

Message Sequence Numbers 355
Metadata Exchanges (Rabbit MQ and Solace) 356
Metadata Exchanges (Tervela) 356
Required Software Components 357
Required Client Configuration 357
Required Appliance Configuration (Rabbit MQ) 357
Required Appliance Configuration (Solace) 357
Required Appliance Configuration (Tervela) 358

Chapter 22 • Using Design Patterns **359**
Overview to Design Patterns 359
Design Pattern That Links a Stateless Model with a Stateful Model 360
Controlling Pattern Window Matches 361
Augmenting Incoming Events with Rolling Statistics 362

Appendix 1 • Supported Functions **365**
Dictionary 367

Appendix 2 • Example: Using a Reserved Word to Obtain an Opcode to Filter Events **417**

Appendix 3 • Example: Using DataFlux Expression Language Global Functions **421**

Appendix 4 • Example: Using Blue Fusion Functions **427**

Appendix 5 • Setting the Logging Level **431**

Recommended Reading **435**
Glossary **437**

About This Book

Audience

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

What's New in SAS Event Stream Processing

Overview

SAS Event Stream Processing 3.1 provides the following enhancements:

- SAS Event Stream Processing Studio
- publish/subscribe security through encryption
- new window types:
  - counter window
  - functional window
  - notification window
  - text category window
  - text sentiment window
- new connectors and adapters:
  - REST
  - sniffer
  - Twitter
- miscellaneous enhancements

SAS Event Stream Processing Studio

SAS Event Stream Processing 3.1 provides a web-based client that enables you to create and test event stream processing models. The client generates XML code based on the models that you create. For more information, see Chapter 4, "Using SAS Event Stream Processing Studio," on page 89.

Publish/Subscribe Security through Encryption

SAS Event Stream Processing 3.1 provides optional encryption for all publish/subscribe clients with the exception of SAS Event Stream Processing Studio. This includes Streamviewer, adapters provided with the product, and any custom adapters.

For more information, see Chapter 19, "Enabling Encryption on Socket Connections," on page 339.

New Window Types

SAS Event Stream Processing 3.1 provides the following new window types:

- Counter windows enables you to determine how many events are streaming through your model and the rate at which they are being processed.
- Functional windows enable you to use different types of functions to manipulate or transform event data.
- Notification windows enable you to send notifications through email, text, or multimedia message.
- Text Category windows generate zero or more categories and associated scores for each event's selected text field.
- Text Sentiment windows report the sentiment of an event (positive, negative, or neutral) with a probability (0 to 1) for each event's text field that is selected.

For more information, see Chapter 2, "Understanding Event Stream Processing Modeling Objects," on page 7.

For more information about counter windows, see Chapter 8, "Creating Counter Windows," on page 141.

For more information about functional windows, see Chapter 9, "Creating Functional Windows," on page 145.

For more information about notification windows, see Chapter 11, "Creating Notification Windows," on page 159.

New Connectors and Adapters

SAS Event Stream Processing 3.1 provides the following new connectors and adapters:

- REST adapter
- sniffer connector and adapter
- Twitter adapter

For more information, see Chapter 16, "Using Connectors," on page 249 and Chapter 17, "Using Adapters," on page 297.

Miscellaneous Enhancements

SAS Event Stream Processing 3.1 provides the following enhancements:

- a caching store for retained data. Use this store for window-retained data that is too large to fit into memory. Using this store can affect performance.
- pattern matching “is” operator.
- new XML tags and various changes to the XML schema:
  - changes to the **`name=`** attribute of XML elements.
  - removal of the **`<esp-server>`** element.
  - XML code validation. The XML server now validates XML models against the schema before starting the model. For more information, see “Validating Your XML Code” on page 36.
- integrated HTTP client driver.

Chapter 1

Overview to SAS Event Stream Processing

Product Overview 1

Getting Started with SAS Event Stream Processing 2
Installing and Configuring SAS Event Stream Processing 2
Using SAS Event Stream Processing 4
Open SAS Event Stream Processing Studio 4
Viewing Logs 4

Writing an Application with SAS Event Stream Processing 5

Product Overview

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

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

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

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

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

processing applications can analyze millions of events per second, with latencies in the milliseconds.

Getting Started with SAS Event Stream Processing

Installing and Configuring SAS Event Stream Processing

SAS Event Stream Processing supports Linux and Microsoft Windows operating environments.

Instructions to configure SAS Event Stream Processing are provided in a ReadMe file that is available in your software depot. This file provides links to system requirements and customer documentation.

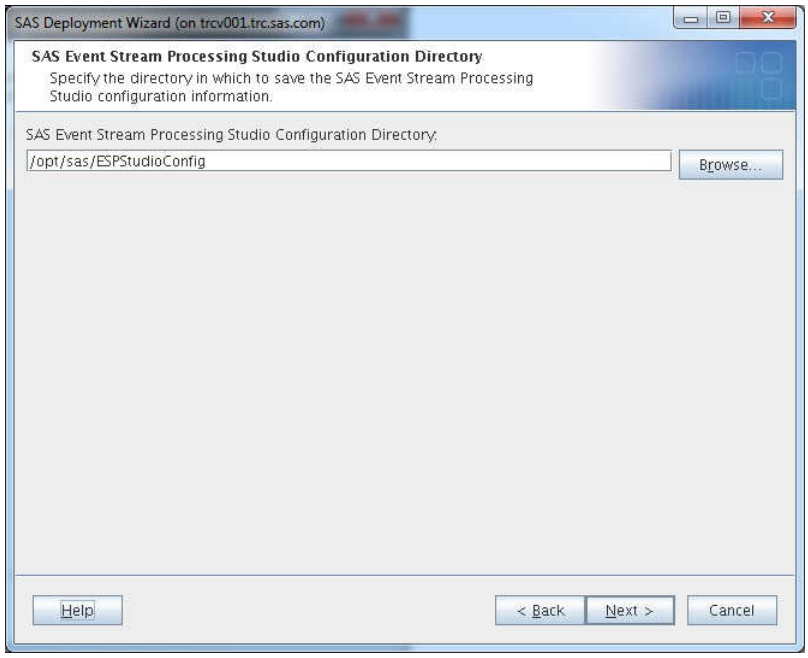
Note: Do not overlay a current release of SAS Event Stream Processing over a previous release. If you want to preserve files from the previous release, rename the associated directories.

Use SAS Deployment Wizard to install SAS Event Stream Processing.

1. Start SAS Deployment Wizard from your SAS Software Depot. On a Linux system, run **`setup.sh`** at the command prompt. On a Microsoft Windows system, run **`setup.exe`**.
2. In the Choose Language dialog box, select the language for the SAS Deployment Wizard. Click **OK**.
3. In the Select Deployment Task step, select **Install SAS Software**. Click **Next**.
4. If you have multiple products in your software depot, select the SAS Event Stream Processing package on the Select SAS Software Order step.

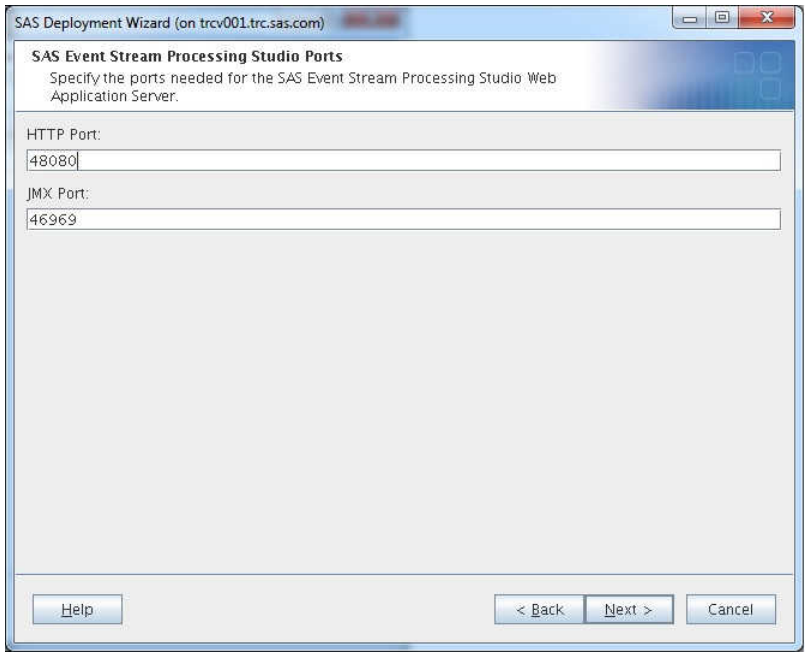
   If SAS Event Stream Processing is the only product in your software depot, this step does not occur.
5. In the Specify SAS Home step, select the **`SASHome`** directory or create a new SAS Home.
6. In the Select Products to Install step, clear all. Select the **SAS Event Stream Processing Engine** and **SAS Event Stream Processing Studio** check boxes. Click **Next**.
7. Depending on your order, you might need to specify the full path to the SAS installation data file and specify regional settings. Click **Next**.
8. In the Configure SAS Event Stream Processing Studio step, leave the **Configure SAS Event Stream Processing Studio** check box selected in order to configure the package. Click **Next**.
9. In the SAS Event Stream Processing Studio Configuration Directory step, specify the directory to use.

Here, the directory **`/opt/sas/ESPStudioConfig`** is specified for a Linux configuration.



Click **Next**.

10. In the SAS Event Stream Processing Studio Ports step, specify the ports to be used by the SAS Web Application Server.



Click **Next**.

11. In the Checking System step, click **Next** when the checking process is complete.
12. In the Deployment Summary step, click **Start**.
13. In the Deployment Complete step, click **Next**.

14. Carefully review the information in the Additional Resources step. When finished, click **Finish** to close the SAS Deployment Wizard.

The **EventStreamProcessingStudioInstructions.html** file explains how to start and stop the SAS Event Stream Processing Studio web application server. It also specifies the URL to use in order to access SAS Event Stream Processing Studio. This file is located in the **documents** subdirectory of the SAS Event Stream Processing Studio configuration directory.

You can use the SAS Deployment Wizard to uninstall SAS Event Stream Processing Studio. Before you do, you must stop running event stream processing processes.

Using SAS Event Stream Processing

After you install SAS Event Stream Processing, you write event stream processing applications using SAS Event Stream Processing Studio, the XML server, or the C++ Modeling API and then execute them. Sample programs in both XML and C++ are available in the **examples** directory of the installation. A readme file in the **examples** directory describes each example and how to run it.

To use SAS Event Stream Processing on Linux hosts, make the following changes to your environment:

- **export DFESP_HOME=!SASROOT/SASEventStreamProcessingEngine/3.1.0**
- **export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$DFEESP_HOME/lib**

To use SAS Event Stream Processing on Microsoft Windows hosts, make the following change to your environment: **DFESP_HOME= PATH=%PATH%;%DFESP_HOME%\bin**.

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

Open SAS Event Stream Processing Studio

After you have started the SAS Web Application Server, you can open SAS Event Stream Processing Studio from a URL with the following format:

http://*esp_studio_hostname*:*port*/SASEventStreamProcessingStudio

If you do not know the host name or port, see the Instructions.html file in your configuration directory.

Viewing Logs

The logs for the SAS Web Application Server are saved in the following directory: ***configuration-directory*/appserver/dfespvm/logs**.

Writing an Application with SAS Event Stream Processing

Use the modeling objects described in Chapter 2, "Understanding Event Stream Processing Modeling Objects," to write an event stream processing application. Follow these steps:

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

   For more information about the XML server, see Chapter 3, "Using the XML Layer," on page 17.

   For more information about the C++ key modeling objects to use to write an application, see Chapter 5, "Programming with the C++ Modeling API," on page 95.

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

   You can publish and subscribe one of several 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, SAS Event Stream Processing Studio, or Streamviewer
   - using the in-process callbacks for subscribe or the inject event block method of continuous queries

   For more information about the publish/subscribe API, see Chapter 15, "Using the Publish/Subscribe API," on page 221.

   Connectors are C++ classes that are instantiated in the same process space as the event stream processor. Connectors can be used from within XML models or C++ models. For more information, see Chapter 16, "Using Connectors," on page 249.

   Adapters use the corresponding connector class to provide stand-alone executables that use the publish/subscribe API. Therefore, they can be networked. For more information, see Chapter 17, "Using Adapters," on page 297.

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

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

Chapter 2

Understanding Event Stream Processing Modeling Objects

Designing an Event Stream Processing Application 7
What Is a Model? 8
Understanding Events 9
Understanding Event Blocks 10
Implementing Engines 11
Understanding Projects 12
Understanding Continuous Queries 12
Understanding Windows 13

Designing an Event Stream Processing Application

SAS Event Stream Processing enables a programmer to write applications that continuously analyze event streams, or events in motion. Conceptually, an event is something that happens at a determinable time that can be recorded as a collection of fields.

When designing an event stream processing application, you must answer the following questions:

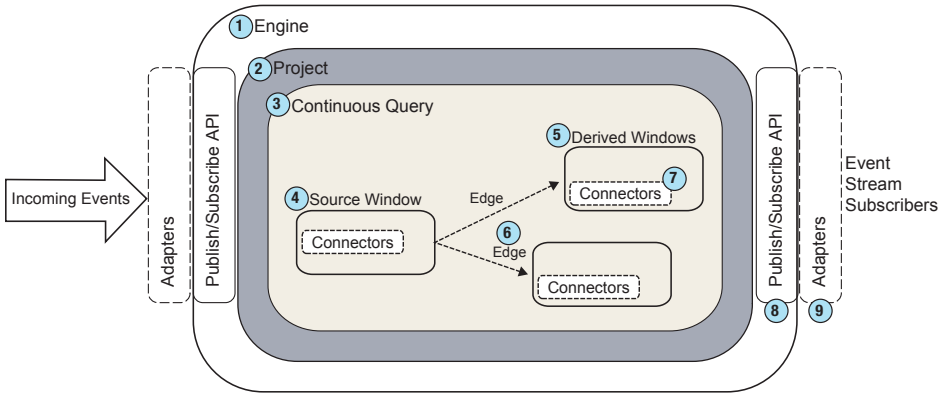
- What event streams are published into an application, and with what protocol and format?
- What happens to the data? That is, how are event streams transformed and analyzed?
- What are the resulting event streams of interest? What applications subscribe these event streams, and in what format and protocol?

Your answers to these questions determine the structure of your model.

What Is a Model?

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

Figure 2.1 *The Event Stream Processing Model Hierarchy*



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

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

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

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

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

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

7 *Connectors* publish or subscribe event streams to and from an engine. Connectors are attached to windows, and provide a lower-level inject call that is in-process to the engine.

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

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

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

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

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

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

Understanding Events

An event creates a packed binary representation of a set of field values. One or more fields of an event must be designated as a primary key. Key fields enable the support of operation codes (opcodes) to enable updates, upserts (for source windows), inserts, and deletes.

Data in an event object is stored in an internal format as described in the schema object. All key values are contiguous and packed at the front of the event. An event object maintains internal hash values based on the key with which it was built. In addition, there are functions in the **`dfESPeventcomp namespace`** for a quick comparison of events created using the same underlying schema.

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

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

The following opcodes are supported by SAS Event Stream Processing:

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

When publishing, 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 or subscribers.

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

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

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

For CSV conversion to binary, see the example application "csv2bin" under the **examples/CXX** directory of the SAS Event Stream Processing 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 client API. You can also convert using the Java client 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.
```

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. Because publish/subscribe operations carry overhead, working with event blocks that contain multiple

events (for example, 512 events per event block) improves throughput performance with minimal impact on latency.

Event blocks can be transactional or normal.

Event Block	Description
Transactional	Processing through the project is atomic. If one event in the event block fails (for example, deleting a non-existing event), then all of the events in the event block fail. Events that fail are logged and placed in an optional bad records file, which can be processed further.
Normal	Processing through the project is not atomic. Events are packaged together for efficiency, but are individually treated once they are injected into a source window.

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

Implementing Engines

Engines are the top level container in the model hierarchy. Each model contains only one engine instance with a unique name. Instantiated engines are processes.

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

- The XML Layer enables you to define single engine definitions and to define an engine with dynamic project creations and deletions. You can use the XML Layer with other products such as SAS Visual Scenario Designer to perform visual modeling.
- You can use SAS Event Stream Processing Studio to create event stream processing models and generate XML code based on those models.
- The C++ Modeling API enables you to embed an event stream processing engine inside an application process space. It also provides low-level functions that enable an application's main thread to interact directly with the engine.

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

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

The decision to implement multiple projects or multiple continuous queries depends on your processing needs. For the XML server, multiple projects can be dynamically introduced, destroyed, stopped, or started because the layer is being used as a service. For all modeling layers, multiple projects can be used for different use cases or to obtain different threading models in a single engine instance. You can use:

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

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

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

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

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

Within a continuous query, you can define a data flow model using all of the available window types. For a description of these window types, see Table 2.1. Procedural windows enable you to write event stream input handlers using C++ or DS2.

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

Understanding Projects

A *project* specifies a container that holds one or more continuous queries and are backed by a thread pool of user-defined size. A project can specify the level of determinism for incremental computations. It can also specify an optional port for publish/subscribe scalability.

The data flow model is always computationally deterministic. 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 engine, each window always processes all data in order. Therefore, data received by a window is never rearranged and processed out of order.

Understanding Continuous Queries

A continuous query specifies a container that holds a directed graph of windows and that enables you to specify the connectivity between windows. Within a continuous query, windows can transform or analyze data, detect patterns, or perform computations. Continuous query processing follows these steps:

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

   If any event in the event block with transactional properties fails, then all of the events in the event block fail. Failed events are logged. They are written to a bad records file for you to review, fix, and republish when you enable this feature.
3. Derived windows transform events into zero or more new events that are based on the properties of each derived window. After new events are computed by derived windows, they flow farther down the model to the next level of connected derived windows, where new events are potentially computed.
4. This process ends for each active path down the model for a given event block when either of the following occurs:

   - There are no more connected derived windows to which generated events can be passed.
   - A derived window along the path has produced zero resulting events for that event block. Therefore, it has nothing to pass to the next set of connected derived windows.

   For information about the XML code that you can use to implement windows, see Chapter 3, “Using the XML Layer,”. For information about the C++ modeling objects to implement windows, see Chapter 5, “Programming with the C++ Modeling API,”.

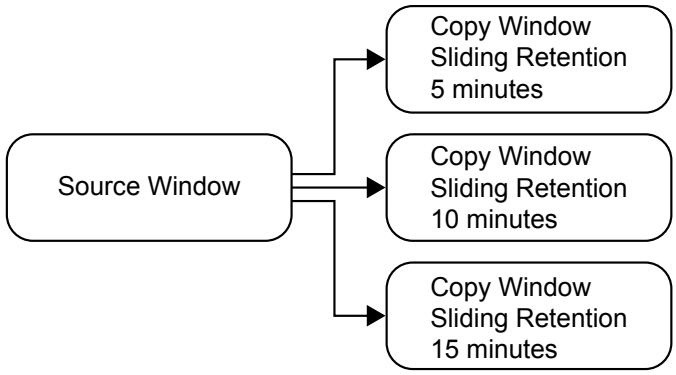
Understanding Windows

A continuous query contains a *source window* and one or more *derived windows*. Windows are connected by *edges*, which have an associated direction.

SAS Event Stream Processing supports the following window types:

Table 2.1 *Window Types Supported by SAS Event Stream Processing*

Window type	Description
Source window	Specifies a source window of a continuous query. All event streams must enter continuous queries by being published or injected into a source window. Event streams cannot be published or injected into any other window type.

Window type	Description
Compute window	Defines a compute window, which enables a one-to-one transformation of input events to output events though the computational manipulation of the input event stream fields. You can use the compute window to project input fields from one event to a new event and to augment the new event with fields that result from a calculation. The set of key fields can be changed within the compute window, but this capability must be used with caution. When a key field change is made within the compute window, the new set of keys must be opcode-compatible with the set of keys from the input streams. That is, Inserts, Updates, and Deletes for the input events' keys must be equivalent Inserts, Update, and Deletes for the new key set.
Copy window	Makes a copy of the parent window. Making a copy can be useful to set new event state retention policies. Retention policies can be set only in source and copy windows. You can set event state retention for a copy window only when the window is not specified to be Insert-only and when the window index is not set to **`pi_EMPTY`**. All subsequent sibling windows are affected by retention management. Events are deleted when they exceed the windows retention policy. 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. 
Aggregate window	An aggregate window is similar to a compute window in that non-key fields are computed. An aggregate window uses the key field or fields for the group-by condition. All unique key field combinations form their own group within the aggregate window. All events with the same key combination are part of the same group. For more information, see Chapter 7, "Creating Aggregate Windows," on page 127.
Counter window	Enables you to see how many events are streaming through your model and the rate at which they are being processed. For more information, see Chapter 8, "Creating Counter Windows," on page 141.
Filter window	Specifies a window with a registered Boolean filter function or expression. This function or expression determines what input events are allowed into the filter window.

Window type	Description
Functional window	Enables you to use different types of functions to manipulate or transform the data in events. Fields in a functional window can be hierarchical, which can be useful for applications such as web analytics. For more information, see Chapter 9, “Creating Functional Windows,” on page 145
Join window	Takes two input windows and a join type. A join window supports equijoins that are one to many, many to one, or many to many. Both inner and outer joins are supported. For more information, see Chapter 10, “Creating Join Windows,” on page 153.
Notification Window	Enables you to send notifications through email, text, or multimedia message. You can create any number of delivery channels to send the notifications. A notification window uses the same underlying language and functions as the functional window. For more information, see Chapter 11, “Creating Notification Windows,”
Pattern window	Enables the detection of events of interest. A pattern defined in this window type is an expression that logically connects declared events of interest. To define a pattern window, you need to define events of interests and then connect these events of interest using operators. The supported operators are "AND", "OR", "FBY", "NOT", "NOTOCCUR", and "IS". The operators can accept optional temporal conditions. For more information, see Chapter 12, “Creating Pattern Windows,” on page 175.
Procedural window	Enables the specification of an arbitrary number of input windows and input handler functions for each input window (that is, event stream). For more information, see Chapter 13, “Creating Procedural Windows,” on page 191.
Text Category window	Enables you to categorize a text field in incoming events. A text category window is Insert-only. The text field could generate zero or more categories with scores. This object enables users who have licensed SAS Contextual Analysis to use its MCO files to initialize a text category window.
Text Context window	Enables the abstraction of classified terms from an unstructured string field. This object enables users who have licensed SAS Contextual Analysis to use its Liti files to initialize a text context window. Use this window type to analyze a string field from an event’s input to find classified terms. Events generated from those terms can be analyzed by other window types. For example, a pattern window could follow a text context window to look for tweet patterns of interest.

Window type	Description
Text Sentiment window	Determines the sentiment of text in the specified incoming text field and the probability of its occurrence. The sentiment value is “positive,” “neutral,” or “negative.” The probability is a value between 0 and 1. A text sentiment window is Insert-only. This object enables users who have licensed SAS Sentiment Analysis to use its SAM files to initialize a text sentiment window.
Union window	Specifies a simple join that merges one or more streams with the same schema. All input windows to a union window must have the same schema. The default value of the strict flag is **`true`**, which means that the key merge from each window must semantically merge cleanly. In that case, you cannot send an Insert event for the same key using two separate input windows of the union window. When the strict flag is set to **`false`**, it loosens the union criteria by replacing all incoming Inserts with Upserts. All incoming Deletes are replaced with safe Deletes. In that case, deletes of a non-existent key fail without generating an error.

Chapter 3

Using the XML Layer

Using the XML Server **18**
Starting the Server 18
Sending HTTP Requests to the XML Server 19

Injecting Events Directly into the XML Server **26**
Overview to XML Server Event Injection 26
Using CSV Format 27
Using XML Format 27
Using JSON Format 28
Injecting Events through HTTP 30

Using the XML Client **31**
Overview to the XML Client 31
Sending GET Requests to the Server with the XML Client 32
Posting Requests to the Server with the XML Client 34

Validating Your XML Code **36**

XML Language Elements **36**
Overview to XML Language Elements 36
XML Language Elements for the Basic Structure of a Model 38
XML Language Elements That Define Window Types 44
XML Language Elements Relevant to Notification Windows 59
XML Language Elements Relevant to Join Windows 70
XML Language Elements for Events 72
XML Language Elements for Connectors 75
XML Language Elements for Functions 77
XML Language Elements Relevant to the HTTP Interface 83

XML Code Examples **84**
Window-Source Example 84
Window-Copy Example 84
Window-Compute Examples 85
Window-Join Examples 86

Using the XML Server

Starting the Server

The XML server is an engine executable that instantiates and executes projects. Each engine contains zero or more projects. The XML server supports control communication through an HTTP interface, through which you can start, stop, create, and remove projects. You can also publish events and query windows. The XML server supports Insert, Update, Upsert, and Delete opcodes.

To start the XML server, use the **$DFESP_HOME/bin/dfesp_xml_server** command.

After starting the server, you can use it in one of two ways:

- execute it as a stand-alone engine with pre-defined projects using the **-model** argument
- run it as a factory server where projects are dynamically created, started, stopped, and destroyed using the **-http-admin** argument

All other command arguments are optional. If you invoke the command without any arguments, you get a usage statement.

Table 3.1 *Command Arguments for the XML Server*

Argument	Description
-model *url*	Specifies the URL to the XML model. Use the URL notation for files, specifying the full path to the XML model (**file:// path**).
-http-admin*port*	Specifies the port on which to receive HTTP administration requests.
– pubsub *n*	Specifies the publish/subscribe port.
-dateformat *f*	Specifies the **strptime**. string
-messages *dir*	Specifies the directory in which to search for localized message files.
-xsd *file*	Specifies the schema file for validation.
-plugindir *dir*	Specifies the directory in which to search for dynamically loaded plug-ins, defaults to **plugins**.
-logconfig *file*	Specifies the SAS logging facility configuration file.
–loglevel esp=*level*	Sets the logging level. Set *level* to **off** | **trace** | **debug** | **info** | **warn** | **error** | **fatal**.

Argument	Description
-http-pubsub ***port***	Sets up a publish/subscribe HTTP server that uses the specified *port*.
-badevents ***file***	Specifies the file to which bad events are written.
-restore ***path***	Specifies the path from which SAS Event Stream Processing restores a project.

Sending HTTP Requests to the XML Server

You can use the following commands to send HTTP requests directly to the XML server. You can send these commands through any client that is capable of formatting an HTTP request, for example:

- a web browser
- the UNIX **curl** command to send HTTP requests to a web server and send responses to the console

You can also send these commands when you invoke **$DFESP_HOME/bin/dfesp_xml_server –http-admin** ***port*** from the command line.

The XML server accepts the following GET requests.

GET Request	Description and Sample Output

`/model?projects=[project1<, project2,... projectN>]&schema=[true | false]`	Retrieve the event stream processing model. You can specify a comma-separated list of project names to retrieve if needed. To include schema information, specify **`schema=true`**.
	`/model?schema=true` can produce
	`<model>`
	`<projects>`
	`<project name='modelingUDFs'>`
	`<contqueries>`
	`<contquery name='cq_01'>`
	`<windows>`
	`<window-source name='src_win'>`
	`<schema>`
	`<fields>`
	`<field key='true' name='ID' type='int32'/>`
	`<field name='city' type='string'/>`
	`<field name='name' type='string'/>`
	`</fields>`
	`<schema-string>name:string,ID*:int32,`
	`city:string</schema-string>`
	`</schema>`
	`</window-source>`
	`<window-compute name='computeWindow'>`
	`<schema>`
	`<fields>`
	`<field key='true' name='ID' type='int32'/>`
	`<field name='city' type='string'/>`
	`<field name='match' type='int32'/>`
	`<field name='name' type='string'/>`
	`<field name='thrice' type='double'/>`
	`<field name='twice' type='int32'/>`
	`</fields>`
	`<schema-string>ID*:int32,name:string,`
	`city:string,match:int32,twice:int32,`
	`thrice:double</schema-string>`
	`</schema>`
	`</window-compute>`
	`</windows>`
	`<edges>`
	`<edge source='src_win' target='computeWindow'/>`
	`</edges>`
	`</contquery>`
	`</contqueries>`
	`</project>`
	`</projects>`
	`</model>`

GET Request	Description and Sample Output

/count	Retrieve event counts for each window in the model.
	/count can produce:
	`<counts>`
	`  <project name="http">`
	`  <contquery name="trades">`
	`  <count window="buys">0</count>`
	`  <count window="buys_aggr">157</count>`
	`  <count window="sells">0</count>`
	`  <count window="sells_aggr">174</count>`
	`  <count window="trades">0</count>`
	`  </contquery>`
	`  </project>`
	`</counts>`

/contents/*project*/*query*/*window*?sort-fields=[*field*]&sort-direction=[ascending | descending]&max-events=[*num*]	Retrieve the events contained in a specified window. You can specify a sort field, a sort direction, and the maximum number of events to return.
	/contents/http/trades/buys_aggr?sort-fields=count&max-event=5 can produce:
	`<contents>`
	`  <results>`
	`  <data opcode="insert">`
	`  <value column="symbol">PFF</value>`
	`  <value column="count">27</value>`
	`  </data>`
	`  <data opcode="insert">`
	`  <value column="symbol">LQD</value>`
	`  <value column="count">24</value>`
	`  </data>`
	`  <data opcode="insert">`
	`  <value column="symbol">PFE</value>`
	`  <value column="count">19</value>`
	`  </data>`
	`  <data opcode="insert">`
	`  <value column="symbol">OIL</value>`
	`  <value column="count">18</value>`
	`  </data>`
	`  <data opcode="insert">`
	`  <value column="symbol">PBW</value>`
	`  <value column="count">17</value>`
	`  </data>`
	`  </results>`
	`</contents>`

/reload	Reload the event stream processing model.
	/reload can produce:
	`<response code='0'>`
	`  <message>esp reloaded</message>`
	`</response>`

GET Request	Description and Sample Output

/persist?path=*directory*	Save the event stream processing model to the specified directory.
	/persist?path=/tmp can produce
	`<response>`
	`  <message>esp successfully saved to '/tmp'</message>`
	`</response>`

/persist/*project*?path=*directory*	Save the specified project to the specified directory.
	/persist/http?path=/tmp can produce:
	`<response>`
	`  <message>project 'http' successfully saved to '/tmp'</message>`
	`</response>`

/restore/*project*?path=*directory*	Restore the specified project from the specified directory.
	/restore/http?path=/tmp can produce:
	`<response>`
	`<message>project 'http' successfully restored from '/tmp'</message>`
	`</response>`

/start-project/*project*	Start the specified project.
	/start-project/http can produce:
	`<response>`
	`  <message>project 'http' successfully started</message>`
	`</response>`

/stop-project/*project*	Stop the specified project
	/stop-project/http can produce:
	`<response>`
	`  <message>project 'http' successfully stopped</message>`
	`</response>`

/delete-project/*project*	Delete the specified project.
	/delete-project/http can produce:
	`<response>`
	`  <message>project 'http' successfully deleted</message>`
	`</response>`

POST Request	Description and Sample Output

/load-*project*?overwrite=[true | false] | start=[true | false] | connectors=[true | false]	Load a project into the server.
	When the **overwrite** parameter is **true**, this command overwrites an existing project of the same name.
	When the **start** parameter is true, this command starts the project.
	When the **connectors** parameter is true, the following command starts the connectors: **/load-project?overwrite=true**
	Input:
	`<project name="http" pubsub="auto" index="pi_EMPTY">`
	`  <contqueries>`
	`  <contquery name="trades">`
	`  <windows>`
	`<window-source name="trades" insert-only="true">`
	`  <schema>`
	`  <fields>`
	`  <field name="id" type="string" key="true"/>`
	`  <field name="symbol" type="string"/>`
	`  <field name="currency" type="int32"/>`
	`  <field name="time" type="int64"/>`
	`  <field name="msecs" type="int32"/>`
	`  <field name="price" type="double"/>`
	`  <field name="quant" type="int32"/>`
	`  <field name="venue" type="int32"/>`
	`  <field name="broker" type="int32"/>`
	`  <field name="buyer" type="int32"/>`
	`  <field name="seller" type="int32"/>`
	`  <field name="buysellflg" type="int32"/>`
	`  </fields>`
	`  </schema>`
	`  </window-source>`
	`<window-filter name="buys">`
	`  <expression><![CDATA[buysellflg==1]]></expression>`
	`  </window-filter>`
	`<window-filter name="sells">`
	`  <expression><![CDATA[buysellflg==0]]></expression>`
	`  </window-filter>`
	`  <window-aggregate name="buys_aggr" index="pi_HASH">`
	`  <schema>`
	`  <fields>`
	`  <field name="symbol" type="string" key="true"/>`
	`  <field name="count" type="int32"/>`
	`  </fields>`
	`  </schema>`
	`  <output>`
	`  <field-expr>ESP_aCount()</field-expr>`
	`  </output>`
	`  </window-aggregate>`

POST Request	Description and Sample Output
/load-*project*?overwrite=[true | false] | start=[true | false] | connectors=[true | false]	Input: (continued) <window-aggregate name="sells_aggr" index="pi_HASH"> <schema> <fields> <field name="symbol" type="string" key="true"/> <field name="count" type="int32"/> </fields> </schema> <output> <field-expr>ESP_aCount()</field-expr> </output> </window-aggregate> </windows> <edges> <edge source="trades" target="buys sells"> <edge source="buys" target="buys_aggr"> <edge source="sells" target="sells_aggr"> </edges> </contquery> </contqueries> </project>
/load-*project*?overwrite=[true | false] | start=[true | false] | connectors=[true | false]	Output: <message>load project 'http' succeeded</message>
/run-*project*	Dynamically load, run, and delete a project. Capture results by specifying windows in the results element: <results> <window key='buys_aggr' sort-fields='count' max-events='3'/> <window key='sells_aggr' sort-fields='count' max-events='3'/> </results>

POST Request	Description and Sample Output
/run-*project*	Input **/run-project**:

```
<request>
 <project name='http' pubsub='none' index='pi_EMPTY'>
    <contqueries>
        <contquery name='trades'>
          <windows>
             <window-source name='trades' insert-only='true'>
                <schema>
                   <fields>
                    <field name='id' type='string' key='true'/>
                    <field name='symbol' type='string'/>
                    <field name='currency' type='int32'/>
                    <field name='time' type='int64'/>
                    <field name='msecs' type='int32'/>
                    <field name='price' type='double'/>
                    <field name='quant' type='int32'/>
                    <field name='venue' type='int32'/>
                    <field name='broker' type='int32'/>
                    <field name='buyer' type='int32'/>
                    <field name='seller' type='int32'/>
                    <field name='buysellflg' type='int32'/>
                   </fields>
                </schema>
             </window-source>
...
</windows>

<edges>
            <edge name='trades'>
              <edge name='buys'>
                 <edge name='buys_aggr'/>
              </edge>
            <edge name='sells'>
                 <edge name='sells_aggr'/>
              </edge>
            </edge>
          </edges>


       </contquery>
    </contqueries>

<event-streams>
      <raw-stream name='tradestream'
       url='file://webroot/files/trades_1K.csv'/>
    </event-streams>
    <prime>
       <inject stream='tradestream' target='trades'/>
    </prime>
 </project>

<results>
    <window key='buys_aggr' sort-fields='count' max-events='3'/>
    <window key='sells_aggr' sort-fields='count' max-events='3'/>
 </results>
</request>
```

POST Request	Description and Sample Output

/run-*project*	Output:
	`<results>`
	`<events window="http/trades/buys_aggr">`
	`<results>`
	`<data opcode="insert">`
	`<value column="symbol">PFF</value>`
	`<value column="count">27</value>`
	`</data>`
	`<data opcode="insert">`
	`<value column="symbol">LQD</value>`
	`<value column="count">24</value>`
	`</data>`
	`<data opcode="insert">`
	`<value column="symbol">PFE</value>`
	`<value column="count">19</value>`
	`</data>`
	`</results>`
	`</events>`
	`<events window="http/trades/sells_aggr">`
	`<results>`
	`<data opcode="insert">`
	`<value column="symbol">LQD</value>`
	`<value column="count">31</value>`
	`</data>`
	`<data opcode="insert">`
	`<value column="symbol">OIL</value>`
	`<value column="count">27</value>`
	`</data>`
	`<data opcode="insert">`
	`<value column="symbol">PFE</value>`
	`<value column="count">24</value>`
	`</data>`
	`</results>`
	`</events>`
	`<message>project http successfully run</message>`
	`</results>`

Injecting Events Directly into the XML Server

Overview to XML Server Event Injection

You can directly inject events into the XML server one of three ways:

- through connectors
- by sending HTTP requests directly to the server
- using the publish/subscribe HTTP interface

You can inject events that have one of the following formats:

- CSV (comma-separated values)
- XML

- JSON

Using CSV Format

Standard CSV event notation, including the opcode, flags, and data, looks like this:

```
i,n,100000,A,87236,1280928600,5048,27.86,488,55000,1012001,5938487,8374755,1
i,n,10035599,LO,87236,1280928600,4172,76,100,55999,1012223,0,8374755,1
i,n,10046471,LOW,87236,1280928600,2005,20.92,1539,55111,1012334,0,8377283,0
i,n,1005973,ANH,87236,1280928600,1707,7.04,235,55333,1012001,5193857,8374755,0
i,n,10093149,LPL,87236,1280928600,1054,15.83,200,55999,101556,0,8359384,1
i,n,10105949,LPX,87236,1280928600,5468,7.15,410,55999,1012223,0,8093853,1
i,n,10105950,LPX,87236,1280928600,728,7.06,100,55000,1012334,0,8093853,0
```

Using XML Format

Using XML format enables you to specify both transactions and individual events. When an element is a transaction element, all event children of that element are injected in a transactional event block. When an element is an event element without a transaction parent, that event is injected in a normal event block.

```
<events>
    <transaction>
        <event>
            <opcode>i</opcode>
            <id key='true'>0</id>
            <symbol>A</symbol>
            <currency>87236</currency>
            <time>1280928600</time>
            <msecs>5048</msecs>
            <price>27.860000</price>
            <quant>488</quant>
            <venue>55000</venue>
            <broker>1012001</broker>
            <buyer>5938487</buyer>
            <seller>8374755</seller>
            <buysellflg>1</buysellflg>
        </event>
        <event>
            <opcode>i</opcode>
            <id key='true'>1</id>
            <symbol>LO</symbol>
            <currency>87236</currency>
            <time>1280928600</time>
            <msecs>4172</msecs>
            <price>76.000000</price>
            <quant>100</quant>
            <venue>55999</venue>
            <broker>1012223</broker>
            <buyer>0</buyer>
            <seller>8374755</seller>
            <buysellflg>1</buysellflg>
        </event>
    </transaction>
  <event>
```

```
    <opcode>i</opcode>
    <id key='true'>2</id>
    <symbol>LOW</symbol>
    <currency>87236</currency>
    <time>1280928600</time>
    <msecs>2005</msecs>
    <price>20.920000</price>
    <quant>1539</quant>
    <venue>55111</venue>
    <broker>1012334</broker>
    <buyer>0</buyer>
    <seller>8377283</seller>
    <buysellflg>0</buysellflg>
  </event>
  <event>
    <opcode>i</opcode>
    <id key='true'>3</id>
    <symbol>ANH</symbol>
    <currency>87236</currency>
    <time>1280928600</time>
    <msecs>1707</msecs>
    <price>7.040000</price>
    <quant>235</quant>
    <venue>55333</venue>
    <broker>1012001</broker>
    <buyer>5193857</buyer>
    <seller>8374755</seller>
    <buysellflg>0</buysellflg>
  </event>
</events>
```

You can use the **<opcode>** element to specify the opcode for the event. The default opcode is Upsert. All other elements specify field names of the schema of the source window into which events are injected.

In the previous example, events with IDs 0 and 1 are injected in a transactional block, and events with IDs 2 and 3 are injected in a normal event block.

Using JSON Format

Using JSON format enables you to specify both transactions and individual events. When a first level JSON entity is an ARRAY, all children of that object are injected in a transactional event block. When an entity is an OBJECT that is not contained in an ARRAY, that event is injected in a normal event block.

```
[
  [
    {
      "opcode": "i",
      "id": 0,
      "symbol": "A",
      "currency": 87236,
      "time": 1280928600,
      "msecs": 5048,
      "price": 27.859999999999999,
      "quant": 488,
      "venue": 55000,
```

```
      "broker": 1012001,
      "buyer": 5938487,
      "seller": 8374755,
      "buysellflg": 1
    },
    {
      "opcode": "i",
      "id": 1,
      "symbol": "LO",
      "currency": 87236,
      "time": 1280928600,
      "msecs": 4172,
      "price": 76.0,
      "quant": 100,
      "venue": 55999,
      "broker": 1012223,
      "buyer": 0,
      "seller": 8374755,
      "buysellflg": 1
    },
  ],
  {
    "opcode": "i",
    "id": 3,
    "symbol": "ANH",
    "currency": 87236,
    "time": 1280928600,
    "msecs": 1707,
    "price": 7.04,
    "quant": 235,
    "venue": 55333,
    "broker": 1012001,
    "buyer": 5193857,
    "seller": 8374755,
    "buysellflg": 0
  },
  {
    "opcode": "i",
    "id": 4,
    "symbol": "LPL",
    "currency": 87236,
    "time": 1280928600,
    "msecs": 1054,
    "price": 15.83,
    "quant": 200,
    "venue": 55999,
    "broker": 101556,
    "buyer": 0,
    "seller": 8359384,
    "buysellflg": 1
  }
]
```


You can use the **`opcode`** property to specify the opcode for the event. The default opcode is Upsert. All other properties specify field names of the schema of the source window into which these events are injected.

In the previous example, events with IDs 0 and 1 are injected in a transactional block, and events with IDs 3 and 4 are injected in a normal event block.

Injecting Events through HTTP

You can use a POST request to send the data directly into the XML server. When you do this, you must set the content-type header of the HTTP request to one of these formats: CSV, XML, or JSON.

The POST request looks like this:

```
http://[server]/inject/[project]/[contquery]/[source window]
            ?blocksize=[n] &quiesce=[true | false]
```

When **quiesce** is true, the project is quiesced before the request returns. The **blocksize** parameter is used to create non-transactional event blocks for injection. Both parameters are optional. The **blocksize** parameter defaults to 1, and the **quiesce** parameter defaults to false. The **quiesce** parameter is not supported by the publish/subscribe HTTP interface.

The GET request looks like this:

```
http://[server]/inject-url/[project]/[contquery]/[source window]
            ?url=[URL] &blocksize=[n] &quiesce=[true | false]
```

The **url** parameter is required. The server determines the content type from the URL. When **quiesce** is true, the project is quiesced before the request returns. The **blocksize** parameter is used to create non-transactional event blocks for injection. Both parameters are optional. The **blocksize** parameter defaults to 1, and the **quiesce** parameter defaults to false. The **quiesce** parameter is not supported by the publish/subscribe HTTP interface.

Note: The POST request assumes a single valid object when processing XML, JSON, or both. However, when the GET command pulls data from a URL, it reads the stream and creates the objects as events arrive. Consider a URL that contains 10 million XML elements. It would be inefficient to create a single XML container with all of these elements and then process them. Thus, the data provided by the specified URL must not have a containing object. In this case, POST XML data should look like this:

```
<events>
        <event>
            <opcode>i</opcode>
            <id key='true'>0</id>
            <symbol>A</symbol>
            <currency>87236</currency>
            <time>1280928600</time>
            <msecs>5048</msecs>
            <price>27.860000</price>
            <quant>488</quant>
            <venue>55000</venue>
            <broker>1012001</broker>
            <buyer>5938487</buyer>
            <seller>8374755</seller>
            <buysellflg>1</buysellflg>
        </event>
        <event>
            <opcode>i</opcode>
            <id key='true'>1</id>
```

```
            <symbol>LO</symbol>
            <currency>87236</currency>
            <time>1280928600</time>
            <msecs>4172</msecs>
            <price>76.000000</price>
            <quant>100</quant>
            <venue>55999</venue>
            <broker>1012223</broker>
            <buyer>0</buyer>
            <seller>8374755</seller>
            <buysellflg>1</buysellflg>
        </event>
</events>
```

A GET request, using a URL that provides the XML, should look like this:

```
<event>
    <opcode>i</opcode>
    <id key='true'>0</id>
    <symbol>A</symbol>
    <currency>87236</currency>
    <time>1280928600</time>
    <msecs>5048</msecs>
    <price>27.860000</price>
    <quant>488</quant>
    <venue>55000</venue>
    <broker>1012001</broker>
    <buyer>5938487</buyer>
    <seller>8374755</seller>
    <buysellflg>1</buysellflg>
</event>
<event>
    <opcode>i</opcode>
    <id key='true'>1</id>
    <symbol>LO</symbol>
    <currency>87236</currency>
    <time>1280928600</time>
    <msecs>4172</msecs>
    <price>76.000000</price>
    <quant>100</quant>
    <venue>55999</venue>
    <broker>1012223</broker>
    <buyer>0</buyer>
    <seller>8374755</seller>
    <buysellflg>1</buysellflg>
</event>
```

Using the XML Client

Overview to the XML Client

Use the command line utility **`dfesp_xml_client`** to communicate with an XML server using the HTTP protocol. You can send GET requests or POST requests as you

need. The client supports communication with the XML server over a secure HTTP channel using HTTPS.

To send an HTTP GET request to the XML server and display the results, use the following command.

dfesp_xml_client -url *URL* <-cert *certificate*> <-pass *passphrase* >

If you are using HTTPS, specify the applicable certificate and pass phrase information.

To send an HTTP POST request to URL with data and display the results.

dfesp_xml_client -url *URL* -post *DATAURL* <-cert *certificate*> <-pass *passphrase* >

Again, if you are using HTTPS, specify the applicable certificate and pass phrase information.

Sending GET Requests to the Server with the XML Client

Get the Current Model

To get the current model from a server named **esp-base**, run the following command line: **dfesp_xml_client -url "http://esp-base:46001/model?schema=true"**

```
http://esp-base:46001/model?schema=true : status 200

<model>
    <projects>
        <project name='project'>
            <contqueries>
                <contquery name='query'>
                    <windows>
                        <window-source name='source'>
                            <schema>
                                <fields>
                                    <field key='true' name='id'
                                     type='string'/>
                                    <field name='data'
                                     type='string'/>
                                </fields>
                                <schema-string>id*:string,data:string
                                </schema-string>
                            </schema>
                        </window-source>
                    </windows>
                    <edges/>
                </contquery>
            </contqueries>
        </project>
    </projects>
</model>
```

Get Event Counts

To get the event counts for windows in the model, run the following command: **dfesp_xml_client -url "http://esp-base:46001/count"**

```
http://esp-base:46001/count : status 200
```

```
<counts>
    <project name='project'>
        <contquery name='query'>
            <count window='source'>4</count>
        </contquery>
    </project>
</counts>
```

Retrieve Events from a Window

To retrieve events from a particular window, run the following command: **dfesp_xml_client -url "http://localhost:46001/contents/project/query/source"**

```
http://localhost:46001/contents/project/query/source : status 200

<contents>
    <results>
        <data opcode='insert'>
            <value column='id'>d98120cb-1cc4-43f7-aed6-5dce5a1acdf5</value>
            <value column='data'>event 4</value>
        </data>
        <data opcode='insert'>
            <value column='id'>010c5888-43a8-41a3-a59b-a7e2f085979f</value>
            <value column='data'>event 3</value>
        </data>
        <data opcode='insert'>
            <value column='id'>5520e8a4-a8b6-432a-9a72-c2e8c7b8dc0c</value>
            <value column='data'>event 2</value>
        </data>
        <data opcode='insert'>
            <value column='id'>8f986c57-0b5c-4e82-b9d0-f7a5e3470c5c</value>
            <value column='data'>event 1</value>
        </data>
    </results>
</contents>
```

Reload the Model

To reload the model, run the following command: **dfesp_xml_client -url "http://localhost:46001/reload"**

```
http://localhost:46001/reload : status 200
<response code='0'>
    <message>esp reloaded</message>
</response>
```

Persist the Model

To persist the model, run the following command: **dfesp_xml_client -url "http://esp-base:46001/persist?path=persistdir"**

```
http://esp-base:46001/persist?path=persistdir : status 200

    esp successfully saved to 'persistdir'
```

Start a Project

To start a project, run the following command: **dfesp_xml_client -url "http://esp-base:46001/start-project/project"**

```
http://esp-base:46001/start-project/project : status 200
<response>
    <message>project 'project' successfully started</message>
</response>
```

Stop a Project

To stop a project, run the following command: **dfesp_xml_client -url "http://esp-base:46001/stop-project/project"**

```
http://esp-base:46001/stop-project/project : status 200
<response>
    <message>project 'project' successfully stopped</message>
</response>
```

Delete a Project

To delete a project, run the following command: **dfesp_xml_client -url "http://esp-base:46001/delete-project/project"**

```
http://esp-base:46001/delete-project/project : status 200
<response>
    <message>project 'project' successfully deleted</message>
</response>
```

Posting Requests to the Server with the XML Client

Overview

To send a POST request to the server, specify the appropriate URL as a parameter along with a URL to the data that you want to send.

Load a Project

To load a project from an XML server named **esp-base**, run the following command: **dfesp_xml_client -url "http://esp-base:46001/load-project?overwrite=true" -post "file://newmodel.xml"**

```
http://esp-base:46001/load-project : status 200

<message>load project 'project' succeeded</message>
```

Inject Events into a Source Window

To inject events into a source window, run the following command: **dfesp_xml_client -url "http://esp-base:46001/inject/project/query/source" -post "file://data.csv"**

```
http://esp-base:46001/inject/project/query/source : status 200
<response>
    <msg>3 events injected into source window project/query/source
         with block size 1</msg>
</response>
```

Run a Project and Get Results

To run a project and get results, run the following command: **`dfesp_xml_client -url "http://esp-base:46001/run-project" -post "file://models/run.xml"`**

```
http://esp-base:46001/run-project : status 200
<results>
    <events window='project/query/brokerAlertsAggr'>
        <results>
            <data opcode='insert'>
                <value column='brokerName'>Lisa</value>
                <value column='frontRunningBuy'>0</value>
                <value column='frontRunningSell'>0</value>
                <value column='openMarking'>23</value>
                <value column='closeMarking'>3</value>
                <value column='restrictedTrades'>3</value>
                <value column='total'>29</value>
            </data>
            <data opcode='insert'>
                <value column='brokerName'>Sally</value>
                <value column='frontRunningBuy'>0</value>
                <value column='frontRunningSell'>1</value>
                <value column='openMarking'>15</value>
                <value column='closeMarking'>3</value>
                <value column='restrictedTrades'>3</value>
                <value column='total'>22</value>
            </data>
            <data opcode='insert'>
                <value column='brokerName'>Joe</value>
                <value column='frontRunningBuy'>1</value>
                <value column='frontRunningSell'>1</value>
                <value column='openMarking'>17</value>
                <value column='closeMarking'>0</value>
                <value column='restrictedTrades'>2</value>
                <value column='total'>21</value>
            </data>
            <data opcode='insert'>
                <value column='brokerName'>Curt</value>
                <value column='frontRunningBuy'>1</value>
                <value column='frontRunningSell'>1</value>
                <value column='openMarking'>21</value>
                <value column='closeMarking'>2</value>
                <value column='restrictedTrades'>2</value>
                <value column='total'>27</value>
            </data>
            <data opcode='insert'>
                <value column='brokerName'>Steve</value>
                <value column='frontRunningBuy'>0</value>
                <value column='frontRunningSell'>1</value>
                <value column='openMarking'>18</value>
                <value column='closeMarking'>3</value>
                <value column='restrictedTrades'>0</value>
                <value column='total'>22</value>
            </data>
        </results>
    </events>
```

```
    <message>project project successfully run</message>
</results>
```

Validate a Model

To validate a project model, run the following command: **dfesp_xml_client -url "http://esp-base:46001/validate-project" -post "file://models/model.xml"**

```
http://esp-base:46001/validate-project : status 200
<schema-validation-success xsd='file:///work/dev/esp/xml/schema/model.rng'/>
```

Validating Your XML Code

To validate XML code that you have written, use the **$DFESP_HOME/bin/dfesp_xml_validate "*XML_file_to_validate*"** command.

The command uses the following to perform a syntactic check on your file:

- the **model.rnc** XML schema definition file that is located in **$DFESP_HOME/etc/xml/schema**
- the **jing.jar** validation code

Note: Do not edit the **model.rnc** XML schema definition file.

When the validation tool finds a violation of the schema definition, it generates error messages that include line numbers and descriptions.

Follow these suggestions to improve the speed of the model creation process and reduce errors:

- Draw the directed graph that represents the continuous query, identifying the window type of each node. At first, make the model insert-only.
- Connect two windows with an edge. Start the XML server and submit the model to the server.
- Use Streamviewer to publish to and subscribe from windows of the query. This enables you to see whether you are getting the expected data flow.
- Successively add windows to the query, each time testing whether the data flow is as you expected.

XML Language Elements

Overview to XML Language Elements

The basic syntax of an XML model for an event stream processing engine is as follows:

```
<engine>
    <projects>
       +<project>
           <contqueries>
               +<contquery>
                   <windows>
```

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

The following set of tables lists the XML language elements that you can use to build models. For each element listed, the following information is provided:

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

XML Language Elements for the Basic Structure of a Model

Table 3.2 *XML Language Elements for the Basic Structure of a Model*

Element	Details	
engine	Description:	The global wrapper for an event stream processing engine.
	Elements:	**projects** [**http-servers**]
	Attributes:	**name=***string* Engine name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**port=**] The publish/subscribe port for the engine [**dateformat**=] The date format to use when converting dates to or from an internal representation. [**on-project-fatal="exit"** | **"exit-with-core"** | **"stop-and-remove"]** Specify how the engine reacts to fatal errors in project. Do one of the following: • Exit with the engine process. • Exit and generate a core file for debugging, or stop all processing. • Disconnect publish/subscribe, clean up all threads and memory, and remove the process from the engine, leaving the engine up and processing other projects. .

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

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

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

Element	Details	
project	Description:	The primary unit in the XML server. Contains execution and connectivity attributes and a list of continuous queries to execute. The thread pool size is specified at the project level. Projects have optional ports for bandwidth distribution.
	Elements:	**contqueries** **[project-connectors]**
	Attributes:	**name=***string* Project name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **threads=***int* A positive integer for the number of threads to use from the available thread pool. **pubsub= none** | **auto** | **manual** Publish/subscribe mode for the project. [**port=***port*] The project-level publish/subscribe port for auto or manual mode [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] A default index type for all windows in the project that do not explicitly specify an index type. [**use-tagged-token= true** | **false**] Specify whether tagged token data flow semantics should be used for continuous queries. [**retention-tracking= true** | **false**] Specify whether to use retention tracking for the project. This allows two events per key in output event blocks. [**disk-store-path=***string*] Specify the path for on-disk event storage.

	Example:	`<project name='analysis' threads='16'`
		`pubsub='manual' port='31417'`
		`index='pi_HASH' use-tagged-token='true'>`
		`<contqueries>`
		`...`
		`</contqueries>`
		`</project>`

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

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

contquery	Description:	The definition of a continuous query, which includes windows and edges. You can use this container to break large models into smaller modules. You can use a project connector or adapter to send event streams from one continuous query to a source window in another.
	Elements	**windows** [**edges**]
	Attributes:	**name=*string*** The name of the continuous query. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**trace=*string***] One or more space-separated window names or IDs. [**index= pi_RBTREE** | **pi_HASH** | **ph_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] A default index type for all windows in the continuous query that do not explicitly specify an index type. **[timing-threshold=*value*]** When a window in the query takes more than *value* microseconds to compute for a given event or event block, a warning message is logged. **[include-singletons=true** | **false]** Specify whether to add unattached source windows.

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

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

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

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

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

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

XML Language Elements That Define Window Types

Table 3.3 XML Language Elements That Define Window Types

Element	Details	
window-aggregate	Description:	An aggregation window.
	Elements:	[**splitter-expr**] |
		[**spliter-plug**]
		schema | **schema-string**
		output
		[**expr-initialize**]
		[**connectors**]
	Attributes:	**name="*string*"**
		Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
		[**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**]
		Index type for the window.
		pubsub= true | **false**
		Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it.
		[**pubsub-index=*value***]
		Publish/subscribe index value.
		[**insert-only**]
		[**output-insert-only**]
		[**collapse-updates**]
		When **true**, multiple update blocks are collapsed into a single update block.
	Example:	See "XML Examples of Aggregate Windows" on page 138.

Element	Details	
window-compute	Description:	A compute window.
	Elements:	[**splitter-expr**] |
		[**spliter-plug**]
		schema | **schema-string**
		output
		[**expr-initialize**]
		[**context-plugin**]
		[**connectors**]
	Attributes:	**name="*string*"**
		Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
		[**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**]
		Index type for the window.
		pubsub= true | **false**
		Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it.
		[**pubsub-index=*value***]
		Publish/subscribe index value.
		[**insert-only**]
		[**output-insert-only**]
		[**collapse-updates**]
		When **true**, multiple update blocks are collapsed into a single update block.
	Example:	See "Window-Compute Examples" on page 85.

Element	Details	
context-plugin	Description:	Specify a function that returns derived context to use with plug-ins.
	Attributes:	**name="*string*"** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **function** The system looks in **plugins/*name*** for the specified ***function*** in the shared object or DLL. It calls that ***function*** that must return a pointer to a C++ class whose base class must be **dfESPcontext**.
	Example:	`<context-plugin name="p1" function="f1"/>`
window-counter	Description:	A counter window.
	Attributes:	**name="*string*"** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**count-interval='*period*'**] [**clear-interval='*cperiod*'**]
	Examples:	See Chapter 8, "Creating Counter Windows," on page 141.

Element	Details	
window-copy	Description:	A copy window.
	Elements:	[**splitter-expr**] |
		[**spliter-plug**]
		[**retention**]
		[**connectors**]
	Attributes:	**name="*string*"**
		Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
		[**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**]
		Index type for the window.
		pubsub= true | **false**
		Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it.
		[**pubsub-index=*value***]
		Publish/subscribe index value.
		[**insert-only**]
		[**output-insert-only**]
		[**collapse-updates**]
		When **true**, multiple update blocks are collapsed into a single update block.
	Example:	See "Window-Copy Example" on page 84.

Element	Details	
window-filter	Description:	A filter window.
	Elements:	[**splitter-expr**] | [**spliter-plug**] **expression** | **plugin** [**expr-initialize**] [**connectors**]
	Attributes:	**name="*string*"** Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= true** | **false** Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] When **true**, multiple update blocks are collapsed into a single update block.
	Example:	`<window-filter name='filterSrc' index='pi_EMPTY'> <expression> not(isnull(src) or isnull(customerURI)) </expression> </window-filter>`

Element	Details	
`window-functional`	Description:	A functional window.
	Elements:	**`schema`** defines the output schema. **`function-context`** defines the entities used to run functions and generate values in the output event. [**`generate`**] contains a function to run that determines whether an output event should be generated from an input event. [**`event-loops`**] enables you to define ways to generate multiple output events from a single input event
	Attributes:	**`name="`*`string`*`"`** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
	Examples:	See Chapter 9, "Creating Functional Windows," on page 145.

Element	Details	
window-join	Description:	A join window.
	Elements:	[**splitter-expr**] |
		[**spliter-plug**]
		join
		output
		[**expr-initialize**]
		[**connectors**]
	Attributes:	**name="*string*"**
		Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
		[**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**]
		Index type for the window.
		pubsub= true | false
		Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it.
		[**pubsub-index=*value***]
		Publish/subscribe index value.
		[**insert-only**]
		[**output-insert-only**]
		[**collapse-updates**]
		When **true**, multiple update blocks are collapsed into a single update block.
		[left-index = pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY| pi_HLEVELDB]
		[right-index = pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY | pi_HLEVELDB]
		Optional overrides for left and right index types.
		Use the **pi_HLEVELDB** to use the on-disk store.
	Example:	See "Window-Join Examples" on page 86.

Element	Details	
window-pattern	Description:	A pattern window.
	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** **patterns** [**connectors**]
	Attributes:	**name="*string*"** Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= true** | **false** Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] When **true**, multiple update blocks are collapsed into a single update block.
	Examples:	See "XML Pattern Window Examples" on page 187.
patterns	Description:	A container for a list of **pattern** elements.
	Elements:	One or more **pattern** elements.
	Attributes:	None.
	Example:	See "XML Pattern Window Examples" on page 187.

Element	Details	
pattern	Description:	A wrapper within a pattern window where events of interest (EOIs), controlling attributes, pattern logic expression, and pattern output rules are grouped.
	Elements:	**events**
	Attributes:	[**index=*fields***] A comma-separated list of fields from the input windows that forms an index generation function.
	Example:	See "XML Pattern Window Examples" on page 187.
logic	Description:	In a **pattern** element, a text string that represents the logical operator expression to combine events of interest (EOIs).
	Elements:	None.
	Attributes:	Enclosed text specifies an expression that defines the pattern. For example: **fby(e1,fby(e2,not(e3)),e4,e5,e6)** when e1, e2, e3, e4, and e5 are named event elements. You can add a temporal condition by appending it to the operator with braces. For example:**fby{30 seconds}**
	Example:	See "XML Pattern Window Examples" on page 187.

Element	Details	
window-procedural	Description:	A procedural window.
	Elements:	[**splitter-expr**] | [**spliter-plug**] **schema** | **schema-string** [**connectors**] [**context-plugin**] [**plugin**] [**ds2-code**]
	Attributes:	**name="*string*"** Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= true** | **false** Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] When **true**, multiple update blocks are collapsed into a single update block.
	Example:	See "XML Examples of Procedural Windows" on page 198.
ds2-code	Description:	A block of DS2 code that is used as an input handler for procedural windows.
	Elements:	CDATA that provides a block of DS2 source code to serve as the handler function.
	Attributes:	**source=*string*** Name of the input widow for this input handler.
	Example:	See "XML Examples of Procedural Windows" on page 198.

Element	Details	
context-plugin	Description:	Specify a function that returns derived context to use with plug-ins.
	Attributes:	**name=*"string"*** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. ***function*** The system looks in **plugins/*name*** for the specified ***function*** in the shared object or DLL. It calls that ***function*** that must return a pointer to a C++ class whose base class must be **dfESPcontext**. <**prop_coll_t**> Specify an optional property list.

	Example:	`<context-plugin`
		`name="p2"`
		`function="f2"`
		`prop_coll_t="prop1, prop2"/>`

Element	Details	
window-source	Description:	A source window.
	Elements:	[**splitter-expr**] |
		[**spliter-plug**]
		schema | **schema-string**
		[**retention**]
		[**connectors**]
	Attributes:	**name="*string*"**
		Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
		[**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**]
		Index type for the window.
		pubsub= true | **false**
		Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it.
		[**pubsub-index=*value***]
		Publish/subscribe index value.
		[**insert-only**]
		[**output-insert-only**]
		[**collapse-updates**]
		When **true**, multiple update blocks are collapsed into a single update block.
		autogen-key={true | **false}**
		Auto-generate the key. The source window must be insert-only and have a single INT64 or STRING key.
	Example:	See "Window-Source Example" on page 84.

Element	Details	
window-textcontext	Description:	A window that enables the abstraction of classified terms from an unstructured string field.
	Elements:	[**splitter-expr**] | [**spliter-plug**] [**connectors**]
	Attributes:	**name="*string*"** Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**index= pi_RBTREE | pi_HASH | pi_LN_HASH | pi_CL_HASH | pi_FW_HASH | pi_EMPTY**] Index type for the window. **pubsub= true | false** Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] When **true**, multiple update blocks are collapsed into a single update block. **liti-fields="*list*"** Comma-separated list of LITI files. **text-field="*fieldname*"** Name of the field in the input window that contains the text to analyze.
	Example:	**<window-textcontext name='TWEET_ANALYSIS' liti-files='/opt/liti/english_tweets.liti,/opt/liti/cs_slang.liti' text-field='tweetBody>**

Element	Details	
window-textsentiment	Description:	A text sentiment window.
	Attributes:	**name="*string*"** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **sam-file** Specify the full path to the SAM file. You must have a SAS Text Analytics license for this to run properly. **text-field** name for the string field in the input event to analyze **index** primary index.

	Example:	`<window-textsentiment name="sentiment"`
		`sam-file="/users/henri/samFile/indep.sam"`
		`text-field="msg"`
		`index="pi_EMPTY"/>`

Element	Details	
window-union	Description:	A union window.
	Elements:	[**splitter-expr**] | [**spliter-plug**] [**connectors**]
	Attributes:	**name="*string*"** Window name. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. [**index= pi_RBTREE** | **pi_HASH** | **pi_LN_HASH** | **pi_CL_HASH** | **pi_FW_HASH** | **pi_EMPTY**] Index type for the window. **pubsub= true** | **false** Publish/subscribe mode for the window. When the project-level value of **pubsub** is **manual**, **true** enables publishing and subscribing for the window and **false** disables it. [**pubsub-index=*value***] Publish/subscribe index value. [**insert-only**] [**output-insert-only**] [**collapse-updates**] When **true**, multiple update blocks are collapsed into a single update block. [**strict = true** | **false**] When true, two inserts on the same key from different inputs fail. When false, all input events that are Inserts are treated as Upserts. In this case, two Inserts for the same key on different input windows work, but you cannot determine which one is applied first.
	Example:	**<window-union name='combinedElementAttributeStats' strict='false' pubsub='true'> </window-union>**

XML Language Elements Relevant to Notification Windows

Table 3.4 *XML Language Elements Relevant to Notification Windows*

Element	Details	
window-notification	Description:	A notification window.
	Elements:	**schema** **function-context** **smtp** **delivery-channels**
	Attributes:	**name="*string*"** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. specifies the window name
	Example:	See "Examples of Notification Windows".
smtp	Description:	Specify the attributes of the Simple Mail Transfer Protocol (SMTP) server to use for email notifications.
	Attributes:	**host='*host*'** Fully qualified host name (for example, **smtp-server.orion.com**) [**user='*username*'**] Fully qualified user name (for example, **henri@orion.com**) [**password='*password*'**] Host password. [**port='*port*'**] Host port.
	Examples:	See "Examples of Notification Windows".
delivery-channels	Description:	A container of elements that specify the delivery channels for a notification window.
	Elements:	**email** **sms** **mms**
	Example:	See "Examples of Notification Windows".

Element	Details	
email	Description:	A container of elements that specify an email notification.
	Elements:	[**deliver**] **email-info** **email-contents**
	Attributes:	**throttle-interval** specifies a time period in which at most one notification is sent to a recipient. **test** a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

	Example:	`<email throttle-interval='2' test='true | false'>`
		`  <deliver>`*code*`</deliver>`
		`  <email-info>`
		`  <sender>`*code*`</sender>`
		`  <recipients>`*code*`</recipients>`
		`  <subject>`*code*`</subject>`
		`  <from>`*code*`</from>`
		`  <to>`*code*`</to>`
		`  </email-info>`
		`  <email-contents>`
		`  <text-content name='blue'>...</text-content>`
		`  <html-content name='red'>...</html-content>`
		`  <image-content name='green'>...</image-content>`
		`  ...`
		`  </email-contents>`
		`</email>`

deliver	Description:	Contains a function to run in order to determine whether a notification should be sent.
	Attributes:	Encloses text that specifies the function to run.
	Example:	`<deliver>` *code* `</deliver>`

Element	Details	
email-info	Description:	A container of elements that represent data to be used to send an email notification.
	Elements:	**sender** **recipients** **subject** **from** **to**

	Example:	`<email-info>`
		`<sender>`*code*`</sender>`
		`<recipients>`*code*`</recipients>`
		`<subject>`*code*`</subject>`
		`<from>`*code*`</from>`
		`<to>`*code*`</to>`
		`</email-info>`

sender	Description:	Specifies the sender address for a notification
	Attributes:	Encloses text that specifies the sender address.
	Example:	`<sender>`*code*`</sender>`
recipients	Description:	Specifies the recipients of a notification.
	Attributes:	Encloses text that specifies the recipients of a notification.
	Example:	`<recipients>`*code*`</recipients>`
subject	Description:	Specifies the subject of a notification.
	Attributes:	Encloses text that specifies the subject of the notification.
	Example:	`<subject>`*code*`</subject>`
from	Description:	Specifies the “from” text of a notification.
	Attributes:	Encloses the “from” text.
	Example:	`<from>`*code*`</from>`
to	Description:	Specifies the “to” text of an email notification.
	Attributes:	Encloses the “to” text.
	Example:	`<to>`*code*`</to>`

Element	Details	
e-mail contents	Description:	A container of elements that specify the contents of an email notification.
	Elements:	**text-content** **html-content** **image-content**
	Example:	<email-contents> <text-content name='*name*'>...</text-content> <html-content name='*name*'>...</html-content> <image-content name='*name*'>...</image-content> ... </email-contents>
html-content	Description:	Specifies HTML content for a notification.
	Attributes:	**name** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. Encloses the HTML code for a notification.
	Example:	<html-content name='*name*'>...</html-content>

Element	Details	
sms	Description:	A container of elements that specify an SMS notification.
	Elements:	[**deliver**] **sms-info** **sms-contents**
	Attributes:	**throttle-interval** specifies a time period in which at most one notification is sent to a recipient. **test = true | false** a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

	Example:	`<sms throttle-interval='interval' test='true | false'>`
		`  <deliver>code</deliver>`
		`  <sms-info>`
		`  <sender>code</sender>`
		`  <subject>code</subject>`
		`  <from>code</from>`
		`  <gateway>code</gateway>`
		`  <phone>code</phone>`
		`  </sms-info>`
		`  <sms-contents>`
		`  <text-content name='name'>...</text-content>`
		`  </sms-contents>`
		`</sms>`

sms-info	Description:	A container of elements that specify information about the SMS notification.
	Elements:	**sender** **subject** **from** **gateway** **phone**

	Example:	`<sms-info>`
		`  <sender>code</sender>`
		`  <subject>code</subject>`
		`  <from>code</from>`
		`  <gateway>code</gateway>`
		`  <phone>code</phone>`
		`  </sms-info>`

Element	Details	
gateway	Description:	Specifies the recipient's provider's SMS gateway
	Attributes:	Encloses text that specifies the gateway
	Example:	`<gateway>code</gateway>`
phone	Description:	Specifies the recipient phone number
	Attributes:	Encloses text that specifies the phone number
	Example:	`<phone>code</phone>`
sms-contents	Description:	A container of elements that specify the contents of an SMS notification
	Elements:	**text-content**

	Example:	`<sms-contents>`
		`  <text-content name='name'>...</text-content>`
		`</sms-contents>`

text-content	Description:	Specifies plain text to be provided in a notification.
	Attributes:	**name="*string*"** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. Encloses the text to be provided in the notification.
	Example:	`<text-content name='name'>...</text-content>`

Element	Details	
mms	Description:	A container of elements that specifies an MMS notification.
	Elements:	**deliver** **mms-info** **mms-contents**
	Attributes:	**throttle-interval** specifies a time period in which at most one notification is sent to a recipient. **test= true | false** a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

	Example:	`<mms throttle-interval='interval' test='true | false'>`
		`  <deliver>code</deliver>`
		`  <mms-info>`
		`  <sender>code</sender>`
		`  <subject>code</subject>`
		`  <gateway>code</gateway>`
		`  <phone>code</phone>`
		`  </mms-info>`
		`  <mms-contents>`
		`  <text-content name='name'>...</text-content>`
		`  <image-content name='name'>...</image-content>`
		`  ...`
		`  </mms-contents>`
		`</mms>`

mms-info	Description:	A container of elements that specifies the content of an MMS notification.
	Elements:	**sender** **subject** **gateway** **phone**

	Example:	`<mms-info>`
		`  <sender>code</sender>`
		`  <subject>code</subject>`
		`  <gateway>code</gateway>`
		`  <phone>code</phone>`
		`  </mms-info>`

Element	Details	
image-content	Description:	Specifies a URL that resolves to image data to be included in a notification.
	Attributes:	**name** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. Encloses text that specifies the URL.
	Example:	`<image-content name='name'>...</image-content>`
function-context	Description:	Enables you to define functions to manipulate event data in a notification window.
	Elements:	**expressions** specifies container for expressions **properties** specifies a container for properties to generate with functions. **functions** specifies container for functions
	Example:	See "Examples of Notification Windows".
expressions	Description	Specifies a container for expressions that you use in a **function-context**.
	Elements:	**expression**

	Example:	`<expressions>`
		`  <expression name='getProtocol'>(.*):</expression>`
		`</expressions>`

expression	Description:	Specifies an expression that you compile a single time for **function-context**.
	Attributes:	**name** Name of the expression. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. Each **expression** element encloses text that specifies a valid POSIX regular expression.
	Example:	`<expression name='getProtocol'>(.*):</expression>`

Element	Details	
properties	Description	Specifies a container for properties to generate with functions. Each specified property is used to generate values to extract from incoming events.
	Elements:	**property-map** **property-xml** **property-json** **property-string** **property-list** **property-set**

	Example:	`<properties>`
		`  <property-map name='' outer='' inner=''>`
		`  [code]`
		`  </property-map>`
		`  <property-xml name=''>[code]</property-xml>`
		`  <property-json name=''>[code]</property-json>`
		`  <property-string name=''>[code]</property-string>`
		`  <property-list name='' delimiter=''>`
		`  [code]`
		`  </property-list>`
		`  <property-set name='' delimiter=''>`
		`  [code]`
		`  </property-set>`
		`  ...`
		`</properties>`

property-map	Description:	Executes the function to generate a map of name-value pairs to be used for value lookups by name.
	Attributes:	**name** The name of the property. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **outer** the outer delimiter to use in parsing data **inner** the inner delimiter to use in parsing data **code** encloses text that specifies the function to run to generate the data to be parsed into a name-value map
	Example:	`<property-map name='' outer='' inner=''>[code]</property-map>`

Element	Details	
property-xml	Description:	Executes the function to generate an XML object that can be used for XPath queries.
	Attributes:	**name** The name of the property. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **code** encloses text that specifies the function to run to generate valid XML
	Example:	`<property-xml name=''>[code]</property-xml>`
property-json	Description:	Executes the function to generate a JSON object that can be used for JSON lookups.
	Attributes:	**name** The name of the property. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **code** encloses text that specifies the function to run to generate valid JSON
	Example:	`<property-json name=''>[code]</property-json>`
property-string	Description:	Executes the function to generate a string for general use in functions.
	Attributes:	**name="*string*"** The name of the property. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **code** encloses text that specifies the function to run to generate a string value
	Example:	`<property-string name='prop'>[code]</property-string>`

Element	Details	
property-list	Description:	Executes the function to generate a list of strings to be used for indexed access.
	Attributes:	**name="*string*"** The name of the property. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **delimiter** the delimiter to use when parsing data **code** encloses text that specifies the function to run to generate data to be base into a value list

	Example:	`<property-list name='proplist' delimiter=''>`
		`  [code]`
		`</property-list>`

property-set	Description:	Executes the function to generate a set of strings to be used for value lookups.
	Attributes:	**name="*string*"** The name of the property. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **delimiter** the delimiter to use when parsing data **code** encloses text that specifies the function to run to generate data to be base into a value set

	Example:	`<property-set name='propset' delimiter=''>`
		`  [code]`
		`</property-set>`

XML Language Elements Relevant to Join Windows

Table 3.5 *XML Language Elements Relevant to Join Windows*

Element	Details	
`conditions`	Description:	A list of left/right field match pairs for joins.
	Elements:	**`fields`**
	Attributes:	Enclosed text specifies a list of one or more field elements that specify the equijoin conditions.
	Example:	See "Window-Join Examples" on page 86.
`fields`	Description:	A container for a list of field elements.
	Elements:	One or more **`field`** elements.
		For **`window-join`**: none.
	Attributes:	None.
		For **`window-join`**: **`left=`*`name`***. A field name from the left input table. **`right=`*`name`*** A field name from the right input table.
	Example:	See **`field`**.
		For **`window-join`**, see "Window-Join Examples" on page 86.

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

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

join	Description:	A container in a join window that collects common join attributes and the **conditions** element.
	Elements:	**conditions**
	Attributes:	**type = fullouter** | **leftouter** | **rightouter** | **inner** Join type. [**no-regenerate = true** | **false**] When true, do not regenerate join changes when the dimension table changes. **use-secondary-index = true** | **false** When true, automatically generate and maintain a secondary index to assist table computation when the dimension table changes. *Note:* The first **edge** element that involves the join window is considered the left window, and the second **edge** element of the join window is considered the right window.
	Example:	See "Window-Join Examples" on page 86.

XML Language Elements for Events

Table 3.6 XML Language Elements for Events

Element	Details	
events	Description:	A wrapper for a list of event elements.
	Elements:	One or more **event** elements. **logic** **output** [**timefields**]
	Attributes:	None.
	Example:	See "XML Pattern Window Examples" on page 187.
event	Description:	The definition of an event of interest (EOI) for pattern matching.
	Elements:	None.
	Attributes:	**name="*string*"** User-specified ID for the event or data source. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **source=*name*** Name of the input window from which this event originates. Enclosed text specifies a WHERE clause that is evaluated by the expression engine.
	Example:	See "XML Pattern Window Examples" on page 187.
timefields	Description:	A container of a list of **timefield** elements.
	Elements:	**timefield**
	Attributes:	None.

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

Element	Details	
timefield	Description:	The field and source pair that specifies the field in a window to be used to drive the time (instead of clock time).
	Elements:	None.
	Attributes:	**field=*name*** User-specified name of the time field. **source=*window*** Name of an input window to which to attach this time field.
	Example:	`<timefield field='firstrun' source='win1'/>`
event-loops	Description:	Specify a container for event loops.
	Elements:	**event-loop-json** **event-loop-xml**
	Attributes:	None
	Examples:	See Chapter 9, “Creating Functional Windows,” on page 145.
event-loop-json	Description:	Use specified or referenced JSON to generate an event
	Elements:	**use-json** | **json reference** **json** **function-context**
	Attributes:	**data=’*data*’**
	Examples:	See Chapter 9, “Creating Functional Windows,” on page 145.
event-loop-xml	Description:	Use specified or referenced XML to generate an event
	Elements:	**use-xml** | **xml reference** **xpath** **function-context**
	Attributes:	**data=’*data*’**
	Example:	See Chapter 9, “Creating Functional Windows,” on page 145.

Element	Details	
`use-json`	Description:	Use JSON code in an event loop.
	Elements:	None
	Attributes:	Enclosed text specifies the JSON code to use in the event loop.
	Examples:	See Chapter 9, “Creating Functional Windows,” on page 145.
`use-xml`	Description:	Use XML code in an event loop
	Elements:	None
	Attributes:	Enclosed text specifies the XML code to use in the event loop.
	Examples:	See Chapter 9, “Creating Functional Windows,” on page 145.
`json`	Description:	Specify the JSON expression to use in an event loop.
	Elements:	None
	Attributes:	Enclosed text specifies the JSON expression in order to retrieve zero or more entities during an event loop.
	Example:	See Chapter 9, “Creating Functional Windows,” on page 145.
`xpath`	Description:	Specify an XML expression to use in an event loop.
	Elements:	None
	Attributes:	Enclosed text specifies an XML expression in order to retrieve zero or more entities during an event loop.
	Examples:	See Chapter 9, “Creating Functional Windows,” on page 145.

XML Language Elements for Connectors

Table 3.7 *XML Language Elements for Connectors*

Element	Details	
connectors	Description:	A list of connector elements.
	Elements:	One or more **connector** elements.
	Attributes:	None.
	Example:	See **connector**.
connector	Description:	A publish or subscribe source-sink for events.
	Elements:	**properties** Each **property name=*value*** specified within a **properties** element encloses text that specify a valid connector property value.
	Attributes:	**name = *string*** Name used as a reference by connector groups. **class = fs | db | mq | pi | project | smtp | sol | tdata | tibrv | tva | rmq | sniffer** For more information about these connector classes and valid connector properties, see Chapter 16, "Using Connectors," on page 249. **type= publish | subscribe**
	Example:	<connectors> <connector class='fs' type='publish> <properties> <property name='type> pub</property> <property name='fstype'> syslog</property> <property name='fsname'> data/clean.log.bi</property> <property name='growinginputfile'> true</property> <property name='transactional'> true</property> <property name='blocksize'> 128</property> </properties> </connector> </connectors>

Element	Details	
connector-entry	Description:	Connector within a connector group.
	Attributes:	**name=*string*** Name of the connector entry. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **state = "finished" | "running" | "stopped"**
connector-groups	Description:	A container for connector-group elements.
	Elements:	One or more **connector-group** elements.
connector-group	Description:	A container of connector-entry elements.
	Elements:	One or more **connector-entry** elements.
	Attributes:	**name=*string*** Name of the connector group. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
project-connectors	Description:	A container for connector-orchestration.
	Elements:	One or more **connector-groups** elements. One or more **edge** elements. *Note:* Project-connector edges connect connector-groups, and do not posses a **slot** attribute.

XML Language Elements for Functions

Table 3.8 *XML Language Elements for Functions*

Element	Details	
context-plugin	Description:	A wrapper for a shared library and function name. The function, when called, returns a **dfESPpcontext** for procedural windows and a **dfESPscontext** for compute windows.
	Elements:	None.
	Attributes:	**name=*sharedlib*** The shared library that contains the context generation function. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **function=*name*** The function that when called returns a new derived context for the procedural window's handler routines.
	Example:	See "XML Examples of Procedural Windows" on page 198.
expression	Description:	An interpreted expression in the DataFlux expression languages. Variables are usually fields from events or are variables declared in the **expr-initialize** element.
	Elements:	None.
	Attributes:	Enclosed text is the expression to be processed.
	Example:	`<expression>quantity >= 100</expression>`
expr-initialize	Description:	An initialization expression code block, common to window types that allow the use of expressions.
	Elements:	**initializer** **udfs**
	Example:	See "Window-Compute Examples" on page 85.

Element	Details	
field-expr	Description:	An algebraic expression whose value is assigned to a field.
	Elements:	None.
	Attributes:	Enclosed text specifies the value of aggregate expression. Includes one or more of the following fields: • **ESP_aSum(*fieldName*)** • **ESP_aMax(*fieldName*)** • **ESP_aMin(*fieldName*)** • **ESP_aAve(*fieldName*)** • **ESP_aStd(*fieldName*)** • **ESP_aWAve(*fieldName*, *fieldName*)** • **ESP_aCount(*fieldName*)** • **ESP_aLast(*fieldName*)** • **ESP_aFirst(*fieldName*)** • **ESP_aLastNonDelete(*fieldName*)** • **ESP_aLastOpCode(*fieldName*)** • **ESP_aGUID()** • **ESP_aCountOpCodes(1|2|3)** For more information, see "Aggregate Functions for Aggregate Window Field Calculation Expressions" on page 129.
		For **window-join**, the enclosed text is any valid SAS DataFlux scalar expression that uses input field names and variables from the **expr-initialize** block. Prefix input field names from the left table with **l_** and input field names from the right input table with **r_**.
	Example:	See "Window-Compute Examples" on page 85.
field-plug	Description:	A function in a shared library whose returned value is assigned to a field.
	Elements:	None.
	Attributes:	**plugin=*name*** Name of the shared library. **function=*name*** Name of the function in the shared library. [**additive = true | false**] Defaults to false.
	Example:	See "Window-Compute Examples" on page 85.

Element	Details	
field-selection	Description:	A reference to a field whose value is assigned to another field.
	Elements:	None.
	Attributes:	**name=*ID*** Selected field in the output schema. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **source=*output_window_field*** The *output_window_field* takes the following form:**l_*field_name*** | **r*field_name.***, where **l_** indicates that the field comes from the left window and **r_** indicates that the field comes from the right window.
	Example:	See "Window-Join Examples" on page 86.
function-context	Description:	Defines entities to run functions and generate values in an output event.
	Elements:	**expressions** **properties** **functions**
	Attributes:	None
	Examples:	See Chapter 9, "Creating Functional Windows," on page 145.
function	Description:	Specify a function to execute.
	Elements:	None
	Attributes:	**name=*funcname*** A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. Enclosed text specifies code for the function.
	Examples:	See Chapter 9, "Creating Functional Windows," on page 145.

Element	Details	
generate	Description:	Specify a function to run that determines whether an output event should be generated from an input event.
	Elements:	None
	Attributes:	Enclosed text specifies code for the function.
	Examples:	See Chapter 9, "Creating Functional Windows," on page 145.
initializer	Description:	Information used to initialize the expression engine.
	Elements:	None
	Attributes:	**type = 'int32' | 'int64' | 'double' | 'string' | 'money' |'date' | 'stamp'** Enclosed text specifies the block of code used to initialize the expression engine. Variables in the initialization block can be used in the filter expression.
output	Description:	A wrapper that is used within several window subtypes to specify how window output is generated.
	Elements:	**field-expr** | **field-plug** Specifically for **window-join**: **field-expr** | **field-plug** | **field-selection**
	Attributes:	None.
	Example:	See "Window-Compute Examples" on page 85.
plugin	Description:	A shared library and function name pair that specifies filter window functions and procedural window handlers.
	Elements:	None.
	Attributes:	**name=*shared_lib*** Shared library that contains the specified function. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **function=*name*** The specified function.
	Example:	See "XML Examples of Procedural Windows" on page 198.

Element	Details	
retention	Description:	Specify the retention policy and value of the retention element.
	Elements:	None.
	Attributes:	**type= bytime_jumping** | **bytime_sliding** | **bycount_jumping** | **bycount_sliding** Retention type. Enclosed text specifies the value of the retention element. When specifying a time-based retention policy, specify number of seconds. When specifying a count-based retention policy, specify a maximum row count. [**field=*name***] Name of a field of type datetime or timestamp. Used to drive the time-based retention policies. If not specified, wall clock time is used. *Note:* Clock time is set relative to GMT.
	Example:	`<retention type='bycount_sliding'>4096</retention>`
schema	Description:	A named list of fields.
	Elements:	**fields**
	Attributes:	[**name=*schema_name***] A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
	Example:	See **field**.
schema-string	Description:	Compact notation to specify a schema.
	Elements:	None.
	Attributes:	Enclosed text specifies the schema.

	Example:	`<schema-string>ID*:int32,`
		`sensorName:string,`
		`sensorValue:double`
		`</schema-string>`

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

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

splitter-plug	Description:	An alternative way to specify splitter functions using a shared library and a function call.
	Attributes:	***string*** Name of shared library containing the splitter function. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **function=*function_name***

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

udfs	Description:	A container of a list of **udf** elements.
	Elements:	**udf**
udf	Description:	Information about user-defined functions.
	Attributes:	**name= *string*** Name of the user-defined function. A name must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9. **type = 'int32' | 'int64' | 'double' | 'string' | 'money' |'date' | 'stamp'** Enclosed text specifies the user-defined expression body.

XML Language Elements Relevant to the HTTP Interface

Table 3.9 *XML Language Elements Relevant to the HTTP Interface*

Element	Details	
http-admin-server	Description:	Enables the configuration of the port through which the XML server receives HTTP requests.
	Elements:	<**ssl_t**>
	Attributes:	**port=*portnum*** Specify the port number on which to listen for HTTP administration requests
	Examples:	`<http-admin-server port='46001'/>`
http-pubsub-server	Description:	Enables the configuration of the HTTP publish/subscribe interface.
	Elements:	<**ssl_t**>
	Attributes:	**port=*portnum*** Specify the port number to use for the publish/subscribe HTTP server.
	Examples:	`<http-pubsub-server port='46002'/>`
http-servers	Description:	A container for HTTP server elements.
	Elements	**http-admin-server** | **http-pubsub-server**
	Attributes:	None

	Examples:	`<http-servers>`
		`  <http-admin-server port='46001'/>`
		`  <http-pubsub-server port='46002'/>`
		`</http-servers>`

ssl	Description:	Pass security information in order for HTTP-based connections to be encrypted. *Note:* Models that include this element cannot be imported into SAS Event Stream Processing Studio.
	Attributes:	**certificate=*name*** Name of the SSL certificate **passphrase=*phrase*** SSL passphrase
	Examples:	`<ssl certificate=`*name*`, passphrase=`*phrase*`/>`

XML Code Examples

Window-Source Example

```
<window-source name='logReadings' index='pi_EMPTY'>
<description>
      <![CDATA[
        Illustration of a source window with a single key field
        and a file/socket connector that populates the source
        window from a file in syslog format. The connector is
        set to accommodate a growing file, that is, one that is
        actively being written to.
       ]]>
</description>
   <schema>
      <fields>
         <field name='ID' type='int32' key='true'/>
         <field name='udate' type='date'/>
         <field name='hostname' type='string'/>
         <field name='message' type='string'/>
      </fields>
   </schema>
   <connectors>
       <connector class='fs'>
         <properties>
             <property name='type'>pub</property>
             <property name='fstype'>syslog</property>
             <property name='fsname'>data/2013-10-11-clean.log.bi</property>
             <property name='growinginputfile'>true</property>
             <property name='transactional'>true</property>
             <property name='blocksize'>128</property>
         </properties>
        </connector>
    </connectors>
</window-source>
```

Window-Copy Example

```
<window-copy name='SrcIPCollision'>
   <description>
        <![CDATA[
          Illustration of a copy window that retains the most recent
          4096 events passed into the copy window. The file/socket
          connector subscribes to the copy window, and outputs the
          windows data to the file named "SrcIPCollision.csv"
          ]]>
   </description>
   <retention type='bycount_sliding'>4096</retention>
   <connectors>
      <connector class='fs'>
```

```
        <properties>
           <property name='type'>sub</property>
           <property name='fstype'>csv</property>
           <property name='fsname'>SrcIPCollision.csv</property>
           <property name='snapshot'>true</property>
        </properties>
      </connector>
   </connectors>
</window-copy>
```

Window-Compute Examples

```
<window-compute name='computeWindow'>
  <description>
        <![CDATA[
        Illustration of a compute window that uses an initialization
        expression to declare a static counter, and then uses a
       field expression to increment the counter, and store this
        incrementing counter in the match field.
        ]]>
  </description>
  <expr-initialize>
    <initializer type='int32'>
    integer counter counter=0
    </initializer>
  </expr-initialize>
  <schema>
    <fields>
      <field name='ID' type='int32' key='true'/>
      <field name='name' type='string'/>
      <field name='city' type='string'/>
      <field name='match' type='int32'/>
    </fields>
   </schema>
   <output>
      <field-expr>name</field-expr>
      <field-expr>city</field-expr>
      <field-expr>counter=counter+1 return counter</field-expr>
   </output>
</window-compute>

<window-compute name='compute'>
  <description>
       <![CDATA[
        Illustration of a compute window that uses a compiled C
        function (named "rowid") in the shared library (named
     "libmethod.so" on linux or "libmethod.dll" on windows)
     to compute the contents of the output field "rowid".
        ]]>
  </description>
  <schema>
    <fields>
      <field name='Id' type='int32' key='true'/>
      <field name='rowid' type='int64'/>
    </fields>
   </schema>
```

```
   <output>
      <field-plug plugin='libmethod' function='rowid'/>
   </output>
</window-compute>
```

Window-Join Examples

```
<window-join name='join_w'>
  <description>
       <![CDATA[
        Illustration of a join window that matches events
     on two fields, "element" and "attribute" present
     in both the left and right input windows. The output fields
     are populated via selecting various fields from the left
     and right input windows using the l_<fieldname> and
        r_<fieldname> conventions.
        ]]>
  </description>
  <join type='leftouter'>
    <conditions>
       <fields left='element' right='element'/>
       <fields left='attribute' right='attribute'/>
    </conditions>
  </join>
  <output>
    <field-selection name='ID' source='l_ID'/>
    <field-selection name='element' source='l_element'/>
    <field-selection name='attribute' source='l_attribute'/>
    <field-selection name='value' source='l_value'/>
    <field-selection name='timestamp' source='l_timestamp'/>
    <field-selection name='status' source='l_status'/>
    <field-selection name='switch' source='r_switch'/>
  </output>
</window-join>

<window-join name='AddTraderName'>
  <description>
       <![CDATA[
        Illustration of a join window that matches events on
     the single field, "traderID (in the left input window)"
     and "ID" (in the right input window). The output fields
     are populated via simple algebraic equations from the left
     and right input windows using the l_<fieldname> and
        r_<fieldname> conventions.
        ]]>
  </description>
  <join type='leftouter'>
    <conditions>
      <fields left='traderID' right='ID' />
    </conditions>
  </join>
  <output>
    <field-expr name='security' type='string'>l_security</field-expr>
    <field-expr name='quantity' type='int32'>l_quantity</field-expr>
    <field-expr name='price' type='double'>100.0*l_price</field-expr>
    <field-expr name='traderID' type='int64'>l_traderID</field-expr>
```

```
    <field-expr name='time' type='stamp'>l_time</field-expr>
    <field-expr name='name' type='string'>r_name</field-expr>
  </output>
</window-join>
```

Chapter 4

Using SAS Event Stream Processing Studio

Overview to SAS Event Stream Processing Studio 89
Using SAS Event Stream Processing Studio 89
Example: Creating a Two-Window Model 91

Overview to SAS Event Stream Processing Studio

SAS Event Stream Processing Studio is a web-based client that enables you to create and test event stream processing models. The client generates XML code based on the models that you create. This visual tool shows a model as a data flow diagram, enabling you to see and control how windows relate and flow into one another.

Note: On Linux and Microsoft Windows platforms, it is recommended that you use Google Chrome to open and use SAS Event Stream Processing Studio. For detailed information about browser support, click and select **About** ⇨ **Browsers and Devices**.

Using SAS Event Stream Processing Studio

To use SAS Event Stream Processing Studio:

1. Enter the following URL in your browser:

   `http://`*`host`*`:`*`port`*`/SASEventStreamProcessingStudio`

   The *host* is the system where SAS Event Stream Processing Studio is installed. The *port* is provided to you during product configuration, and is the HTTP port.

   The initial SAS Event Stream Processing Studio window appears.

   If this is the first time you are using SAS Event Stream Processing Studio, the window is blank. If not, models listed are those that have already been created or imported into the modeler.

2. To open an existing model, click the name. The workspace opens, displaying a diagram of the selected model.

3. To create a new model, click **New**. You are prompted to choose a new engine or a new project.

   If you select engine, the workspace opens with an engine that contains one project, which contains one continuous query.

   If you select project, the workspace opens with a project that contains one continuous query.

4. When you first open or create a model, the XML Factory Server window appears. Enter the **Host**, **Admin Port**, and **HTTP Pub/Sub Port** values you used to start the XML server.

   For information about how to start an XML server, see "Starting the Server" on page 18.

   The model appears in the left pane. The **Properties** of the selected modeling object appears in the right pane.

5. Click **Containers** to select and drag continuous queries to a project. Click **Windows** to select and drag windows to a continuous query.

   Alternatively, click and select **Dock Objects Panel** to dock the panel to the left of the model. Drag objects from the docked panel to the model as needed.

6. When you run your model, you can use the **Test** button to see how it runs. Click **Test**. The Test window appears. Click on a window and select **Subscribe**. Event data appears in the **Results** section.

   Note: Test mode is supported only for projects. It is not supported for engines.

   Click **Close** to close the Test window.

7. When you are finished making changes to a model, click to save your changes.

8. Click an object in the workspace and then click to view the XML that is generated. To export the XML, click and select **Export Project XML**.

9. Click to exit a model.

   You are prompted to save any outstanding changes. After responding, you are returned to the SAS Event Stream Processing Studio window.

10. To delete a model, click **Select** on the initial SAS Event Stream Processing window. Select the check box next to a model name and click . The model is removed.

    Click **Cancel** to return to the main window.

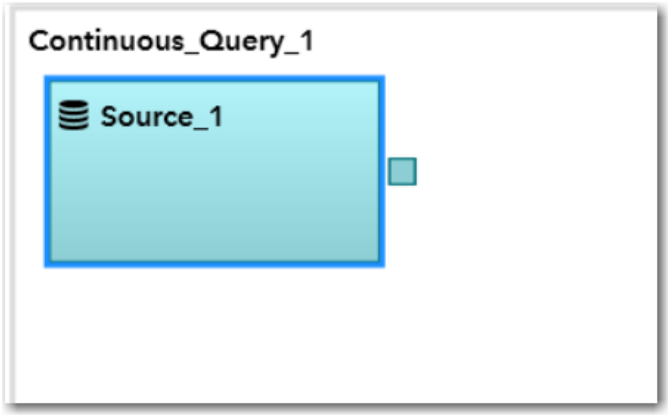
11. To exit SAS Event Stream Processing Studio, close the tab or window in which the application is running.

Use the **Import** button to import an existing XML program into SAS Event Stream Processing Studio.

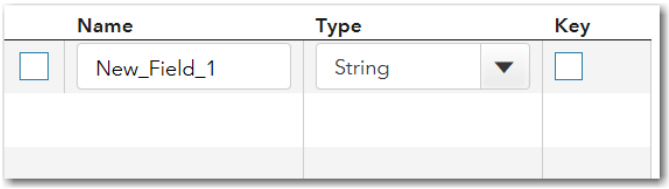
Example: Creating a Two-Window Model

The following example creates a simple model linking a source window to a filter window.

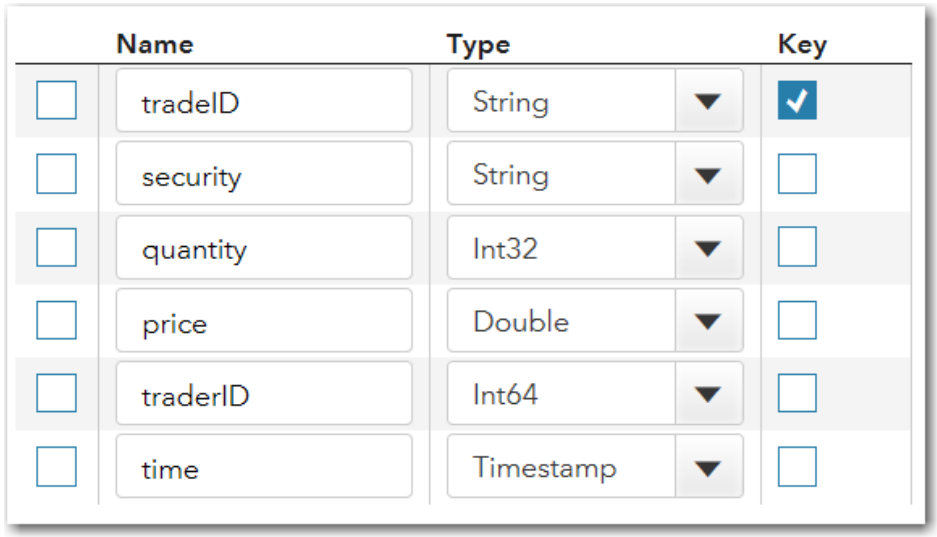
1. Open SAS Event Stream Processing Studio. Click **New**. Select **New Project**.
2. Enter valid values in the **Host**, **Admin Port**, and **HTTP Pub/Sub Port** fields of the XML Factory Server window. A blank project appears with a blank continuous query within it.
3. Click **Windows**, select **Source**, and drag it to the continuous query. The workspace now looks like this.



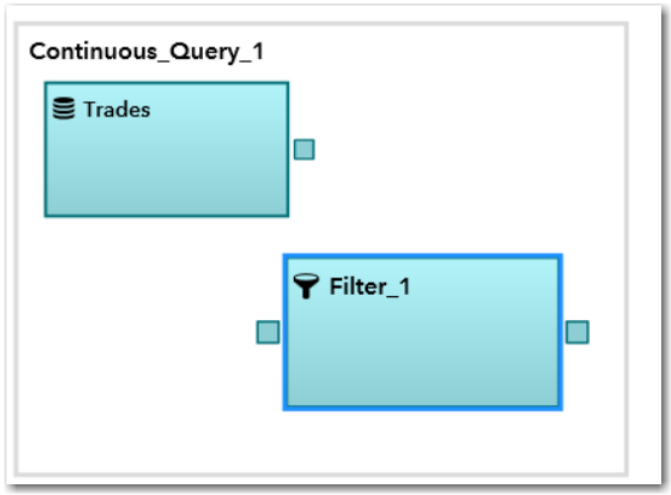
4. Click by the **Name** field on the **Properties** pane to rename the source window. The Rename window appears. Enter `Trades` and click **OK**.
5. Specify schema for the `Trades` window by expanding the section and clicking . You are prompted to enter a new field for the schema. You must specify one field to be the **Key** field.



Click to enter each new field. Enter the following values to specify a schema for this source window:

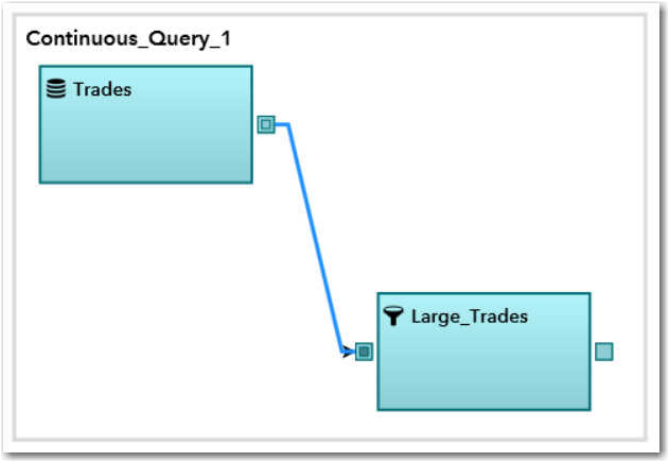


6. Click **Windows**, select **Filter**, and drag it to the continuous query. The workspace now looks like this.



7. Rename the filter window `Large_Trades`.
8. Expand the **Output Fields** section and enter `quantity>=100` in the **Expression** box.

9. Click on the box on the right side of the Trades window to the box. Drag the arrow to the left side of the Large_Trades window.



The filter window now accepts trades from the source window and filters events when the quantity of the trade is greater than or equal to 100.

10. To save the new model, click .

11. To view the XML generated by the model that you created, click . The XML appears to the right of the created model.

XML

```
<project name="Project_2" pubsub="none" threads="1">
 <contqueries>
  <contquery name="Continuous_Query_1">
   <windows>
    <window-source name="Trades" collapse-updates="true">
     <schema>
      <fields>
       <field name="tradeID" type="string" key="true"/>
       <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-source>
    <window-filter name="Large_Trades" collapse-updates="true"
     <expression><![CDATA[quantity >=100]]></expression>
    </window-filter>
   </windows>
   <edges>
    <edge source="Trades" target="Large_Trades"/>
   </edges>
  </contquery>
 </contqueries>
</project>
```

12. Exit SAS Event Stream Processing Studio by closing the tab.

Chapter 5

Programming with the C++ Modeling API

Overview to the C++ Modeling API **95**

Dictionary **96**
dfESPengine 96
dfESPdatavar 97
dfESPschema 98
dfESPevent 99
dfESPeventblock 100
dfESPproject 101
dfESPcontquery 103
dfESPwindow_source 103
dfESPwindow_aggregate 105
dfESPwindow_copy 105
dfESPwindow_compute 106
dfESPwindow_counter 109
dfESPwindow_filter 110
dfESPwindow_functional 112
dfESPwindow_join 112
dfESPwindow_notification 113
dfESPwindow_pattern 114
dfESPwindow_procedural 114
dfESPwindow_textCategory 115
dfESPwindow_textContext 116
dfESPwindow_textSentiment 118
dfESPwindow_union 119
dfESPpersist 119

Overview to the C++ Modeling API

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

Alternatively, you can embed an engine into the process space of either an existing application 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++ modeling objects. For information about how to define models using the XML Modeling Layer, see Chapter 3, "Using the XML Layer," on page 17.

For information about how to control the order in which connectors execute, see "Orchestrating Connectors" on page 252.

Dictionary

dfESPengine

specifies the top-level container or manager in a model.

Syntax

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

Required Arguments

enginePtr
the pointer to the engine

argc
argument count as passed into main

argv
argument vector as passed into main.

Accepts the following options:

- **`-t`** *textfile.name* to write output
- **`-b`** *badevent.name* to write events that failed to be applied to a window index
- **`-r`** *restore.path* to restore a previously persisted engine state.
- **`-h`** *http-pubsub-port* to specify a port for a restful publish/subscribe interface. Use this to access the server from Streamviewer.

pubsub_ENABLE(*port#*) | pubsub_DISABLE
indicates 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 **`dfESPLLInfo`**.

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

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

Details

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

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

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

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

Example

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

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

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

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

/* Now cleanup and shutdown gracefully */

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

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

dfESPdatavar

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

Syntax

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

Required Arguments

datavarPtr
pointer to the variable

data_type
can be one of the following values:

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

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

Example

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

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

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

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

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

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

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

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

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

dfESPschema

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

Syntax

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

Required Arguments

name
user-supplied name of the representation

schema
specifies the structure of fields of event data

Details

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

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

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

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

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

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

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

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

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

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

dfESPevent

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

Syntax

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

Required Arguments

name
user-supplied name of the event

schemaPtr
user-supplied schema pointer

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

i | u | p | d	means Insert, Update, Upsert, and Delete respectively
n | p	means normal event or partial-update event
f1, f2,..., fn	are the fields that make up the data portion of the event

failedEvent
TRUE when the call is successful, and **FALSE** otherwise

Details

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 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.

dfESPeventblock

creates a lightweight wrapper around a collection of events.

Syntax

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

Required Argument

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

Details

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

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

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

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

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

A **`dfESPeventblock`** with more than one event, but without transactional properties set, can be used to improve performance during the absorption of the event block into the appropriate source window. You use this to trade off a little bit of latency for a large gain in throughput. It is best to test the event block optimal size trade-off. For example, placing 256 events into an event block gives both great latency and throughput. This performance varies depending on the width of the events.

dfESPproject

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

Syntax

dfESPproject **projectPtr*;
enginePtr—>newProject(“*projectname*”, <true | false>);

Required Arguments

projectPtr
the pointer to the project

enginePtr
the pointer to the engine as specified through **`dfESPengine`**.

true | false
indicates whether tagged token data flow should be enabled

Details

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

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

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

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();
```

dfESPcontquery

specify a continuous query object.

Syntax

dfESPcontquery **contqueryPtr*;
contqueryPtr = *projectPtr*—> newContQuery(“*queryID*”);

Required Arguments

contqueryPtr
the pointer to the continuous query

queryID
user-supplied ID of the continuous query object

projectPtr
the pointer to the project as specified in **`dfESPproject`**

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.

Syntax

dfESPwindow_source *_windowPtr_;
windowPtr=_contqueryPtr_—> newWindow_Source
dfESPString (“_window_label_”), dfESPIndextypes::_index_,
dfESPstring (“_schema_”);

Required Arguments

windowPtr
the pointer to the source window

contqueryPtr
the pointer to the continuous query object as specified by **`dfESPcontquery`**

index
primary index. Six types of primary indexes are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

schema
user-supplied string that specifies the structure of the fields of event data.

Details

The source window is the only window type to which event streams can be published. All other window types are called derived windows, which transform or analyze event streams coming into them from other windows.

Example

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

```
dfESPwindow_source *sw;
sw = cq->newWindow_source dfESPstring ("mySourceWindow"),
     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.

Under the following circumstances, a source window can auto-generate the key value:

- the source window is Insert only
- there is only one key for the window
- the key type is INT64 or string

When these conditions are met and the **`setAutoGenerateKey()`** call is made, you do not have to supply the key value for the incoming event. The source window overwrites

the value with an automatically generated key value. For INT64 keys, the value is an incremental count (0, 1, 2, ...). For STRING keys, the value is a Globally Unique Identifier (GUID).

dfESPwindow_aggregate

specifies an aggregation window.

Syntax

dfESPwindow_aggregate **windowPtr*;
windowPtr=contqueryPTR—> newWindow_aggregate dfESPstring ("*windowID*"), dfESPIndextypes::*index*, dfESPstring *schema*);

Required Arguments

windowPtr
pointer to the aggregate window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

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 7, "Creating Aggregate Windows," on page 127

dfESPwindow_copy

makes a copy of the parent window.

Syntax

dfESPwindow_copy **windowPtr*;
windowPtr=contqueryPtr—> newWindow_copy
dfESPstring ("*windowID*"), dfESPIndextypes::*index*;

Required Arguments

windowPtr
the pointer to the copy window

contqueryPtr
the pointer to the continuous query object specified in **dfESPcontquery**

windowID
user-supplied identifier of the copy window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

Details

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. You can define retention policies only in source and 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.

dfESPwindow_compute

defines a compute window

Syntax

dfESPwindow_compute **windowPtr*;
windowPtr=contqueryPtr—> newWindow_compute dfESPstring ("*windowID*"), dfESPIndextypes::*index*, dfESPstring *schema*;

Required Arguments

windowPtr
pointer to the compute window

contqueryPtr
the pointer to the continuous query object specified in **dfESPcontquery**

windowID
user-supplied identifier of the compute window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

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

Details

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

Examples

Example 1

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

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

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

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

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

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

Example 2

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

Assume the following schema for input events:

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

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

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

Here is the scalar function:

```
dfESPdatavar *priceBYquant(dfESPschema*is, dfESPevent *nep,
      dfESPevent *oep, dfESPscontext *ctx) {
   //
   // If you are getting an update, then nep is the updated
   //      record, and oep is the old record.
```

```
    //
    // Create a null return value that is of type double.
    //
    dfESPdatavar *ret = new dfESPdatavar(dfESPdatavar::ESP_DOUBLE);
    // If you were called because a delete is being issued, you do not
    //      compute anything new.
    //
    if (nep->getOpcode() == dfESPeventcodes::eo_DELETE)
       return ret;
    void *qPtr = nep->getPtrByIntIndex(2); // internal index of
         quant is 2
    void *pPtr = nep->getPtrByIntIndex(3); // internal index of
         price is 3
    if ((qPtr != NULL) && (pPtr != NULL)) {
       double price;
       memcpy((void *) &price, pPtr, sizeof(double));
       int32_t quant;
       memcpy((void *) &quant, qPtr, sizeof(int32_t));
       ret->setDouble(quant*price);
    }
    return ret;
}
```

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

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

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

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

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

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

Example 3

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

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

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

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

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

// Register as many field expressions as there are non-key
//      fields in the output schema.
cw->addNonKeyFieldCalc("symbol”); // pass through the symbol
     value
cw->addNonKeyFieldCalc("price”); // pass through the price
     value
cw->addNonKeyFieldCalc("price*quantity”); // compute cost
     = price * quantity
```

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

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

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

dfESPwindow_counter

Enables you to see how many events are streaming through your model and the rate at which they are being processed.

Syntax

dfESPwindow_counter **windowPtr*;
windowPtr = *contqueryPtr*->newWindow_counter("*windowID*",dfESPindextypes::*index*);

Required Arguments

windowPtr
pointer to the counter window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied ID of the counter window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

Example

```
dfESPproject     *project = engine->newProject("project");
dfESPcontquery    *contquery = project->newContquery("contquery");
dfESPwindow_source    *source =
  contquery->newWindow_source("source",dfESPindextypes::pi_RBTREE,
                              dfESPstring("ID*:int32,symbol:string,valstr:string"));
dfESPschema *schema = source->getSchema();

dfESPwindow_counter *counter = contquery->newWindow_counter("counterWindow",
                                          dfESPindextypes::pi_RBTREE);
counter->setCountInterval("2 seconds");
counter->setClearInterval("10 seconds");

contquery->addEdge(source,0,counter);
```

dfESPwindow_filter

specifies a filter window

Syntax

dfESPwindow_filter **windowPtr*;
windowPtr=contqueryPtr—> newWindow_filter dfESPstring ("*window_ID*"),
dfESPIndextypes::*index*;
*windowPtr->*setFilter(*filterFunction* | *filterExpression*);

Required Arguments

windowPtr
user-supplied pointer to the filter window

contqueryPtr
user-supplied pointer continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied ID of the filter window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

filterFunction
user-supplied identifier of the filter function

filterExpression
user-supplied identifier of the filter expression

Details

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

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

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

Examples

Example 1

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

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

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

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

     // If field is null, filter always fails.
     //
     if (dv.isNull()) {
          prv->setI32(0); // the return value to 0
     } else {
          // Get the int32 value from the datavar and compare to 1000
          if (dv.getI32() < 1000) {
               prv->setI32(0);     // set return value to 0
          } else {
               prv->setI32(1);     // set return value to 1
          }
     }
     return prv; // return it.
```

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

```
dfESPwindow_filter *fw_01;
     fw_01 = cq->newWindow_filter("filterWindow_01",
                                  dfESPindextypes::pi_RBTREE);
     fw_01->setFilter(booleanScalarFunction);
```

```
// Register the filter UDF.
```

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

Example 2

The following code example uses filter expressions.

```
dfESPwindow_filter *fw_01;
fw_01 = cq->newWindow_filter("filterWindow_01",
                             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_functional

Enables you to use different types of functions to manipulate or transform data in streamed events.

Syntax

dfESPwindow_functional **windowPtr*;
windowPtr= *contqueryPtr* ->newWindow_functional("*windowID*",dfESPindextypes::*index*, "*schema*");

Required Arguments

windowPtr
pointer to the functional window

contqueryPtr
pointer to the continuous query object specified in **dfESPcontquery**

windowID
user-supplied identifier of the functional window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

schema
***id*:int64,*symbol*:string,*timeString*:string,*hourOfDay*:double,*quant*:int64**

Details

For more information, see Chapter 9, "Creating Functional Windows,".

dfESPwindow_join

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

Syntax

dfESPwindow_join **windowPtr*;
windowPtr=contqueryPtr—> newWindow_join dfESPstring (“*windowID*”),
dfESPwindow_join::*jointype*, dfESPIndextypes::*index*;

Required Arguments

windowPtr
pointer to the join window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the join window

window_label
user-supplied description of the window

jointype
type of join to be applied

index
primary index. Six types of primary indexes are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

See Also

Chapter 10, “Creating Join Windows,” on page 153

dfESPwindow_notification

Enables you to send notifications through email, text, or multimedia message. You can create any number of delivery channels to send the notifications.

Syntax

dfESPwindow_notification **windowPtr*;
windowPtr = *contqueryPtr*->newWindow_notification("notify",NULL);

Required Arguments

windowPtr
pointer to the notification window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

Details

For more information, see Chapter 11, “Creating Notification Windows,”.

Example

The following code sample sets up an email notification to a broker.

```
dfESPwindow_notification *notification =
   contquery->newWindow_notification("notify",NULL);
   notification->setSmtpConnection("mailhost.fyi.orion.com");

dfESPemail    *email = notification->addEmail();
email->setThrottleInterval("5 minutes");
email->setSender("john.doe@orion.com");
email->setRecipients("$email");
email->setSubject("Investment Opportunity");
email->setFrom("ESP");
email->setTo("Wealthy Trader");
email->addText("You traded $quant shares of $symbol at $$price.");
email->setTestMode(true);

contquery->addEdge(joinBrokerData,0,notification);
```

dfESPwindow_pattern

enables the detection of events of interest.

Syntax

**dfESPwindow_pattern ***_windowPtr_;
windowPtr=_contqueryPtr_—> newWindow_pattern dfESPstring (“_windowID_”)
, dfESPIndextypes::_index_, dfESPstring _schema_) ;

Required Arguments

windowPtr
pointer to the pattern window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the pattern window

index
primary index. Six types of primary indexes are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

schema
schema of the pattern window

See Also

Chapter 12, “Creating Pattern Windows,” on page 175

dfESPwindow_procedural

specifies a procedural window

Syntax

dfESPwindow_procedural **windowPtr*;
windowPtr= *contqueryPtr*—> newWindow_procedural
dfESPstring (“*windowID*”), dfESPIndextypes::*index*, dfESPstring *schema*;

Required Arguments

windowPtr
pointer to the procedural window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the procedural window

index
primary index. Six types of primary indexes are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

schema
schema of the procedural window defined by **`dfESPstring`**

See Also

Chapter 13, “Creating Procedural Windows,” on page 191

dfESPwindow_textCategory

enables you to categorize text fields in incoming events

Syntax

dfESPwindow_textCategory **windowPtr*;
windowPtr = *contqueryPtr* —> newWindow_textCategory (dfESPstring (“*windowID*”), pindex_t *index*, dfESPstring *mcoFile*, dfESPstring *inputFieldName*);

Required Arguments

windowPtr
pointer to the text category window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the window

index
primary index. Six types of primary indexes are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

mcoFile
path to the MCO file

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

Details

Because this window is insert-only, the following is recommended:

- you follow the window with a copy window so that events can be retained as needed
- you use an empty index for the window

You need a SAS Contextual Analysis license so that you can provide the required *mcoFile* .

The window generates its own schema, which is as follows:
key_fields_of_input_window, **`Category:string, score:double`**.

dfESPwindow_textContext

specifies a text context window.

Syntax

dfESPwindow_textContext **windowPtr*;
windowPtr=contqueryPtr—> newWindow_textContext(dfESPstring (“*windowID*”), dfESPIndextypes::*index*, *litiFiles*, *inputFieldName*);

Required Arguments

windowPtr
pointer to the text context window

contqueryPtr
pointer to the continuous query object that is specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the text context window

index
primary index. Six types of primary index are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

litiFiles
comma separated list of Liti file paths.

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

Details

You need a SAS Contextual Analysis license so that you can provide the required *litiFiles*.

The schema of events output by the **`textContext`** window is as follows:

```
"input event key fields*, termID*:int64, term:string, termLen:int32,
 tag:string, tagLen:int32, locStart:int32, locEnd:int32, wordStart:int32,
wordEnd:int32"
```

Example

The following example uses an empty index for the **textContext** window, which is insert only. That way, the window does not grow endlessly. The **textContext** window is followed by a copy window that uses retention to control the growth of the classified terms.

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

        // Build the textContext window. We specify the window name, the depot
        //    used for retained events, the type of primary index, the Liti
        //    files specification string, and the input string field to analyze.
        // Note that the index for this window is an empty index.  This means
        //    that events created in this window will not be retained in this
        //    window.  This is because textContext windows are insert only,
        //    hence there is no way of deleting these events using retention
        //    so without using an empty index this window would grow indefinitely.
        // If you try to run this example you will need to have licensed SAS
        //    Contextual Analysis, whose install contains these Liti language files.
        // You will need to change the litFiles string below to point to your
        //    installation of the Liti files otherwise the text analytics engine
        //    will not be initialized and classified terms will not be found.
        //
        dfESPwindow_textContext *tcw_01;

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

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

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

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

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

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

Here are the results.

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

dfESPwindow_textSentiment

generates a new event for every incoming event with the incoming event's key fields, sentiment value, and probability. The sentiment value is "positive," "neutral," or "negative." The probability is a value between 0 and 1.

Syntax

dfESPwindow_textSentiment **windowPtr*;
windowPtr =
contqueryPtr—> newWindow_textSentiment(dfESPstring ("*windowID*"), dfESPindextype::*index*, dfESPstring *samFile*, dfESPstring *inputFieldName*);

Required Arguments

windowPtr
pointer to the text sentiment window

contqueryPtr
pointer to the continuous query object specified in **dfESPcontquery**

windowID
user-supplied ID of the text sentiment window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" on page 208.

samFile
path to the SAM file

inputFieldName
user-supplied name of the string field in the input event to analyze

Details

Because this window is insert-only, the following is recommended:

- you follow the window with a copy window so that events can be retained as needed
- you use an empty index for the window

You need a SAS Sentiment Analysis license so that you can provide the required *samFile* .

The window generates its own schema, which is as follows: ***key_fields_of_input_window,*** **`sentiment:string, probability:double`**.

dfESPwindow_union

specifies a union window

Syntax

dfESPwindow_union **windowPtr*;
windowPtr=contqueryPtr—> newWindow_union(dfESPstring (“*windowID*”), dfESPIndextypes::*index*, true | false);

Required Arguments

windowPtr
pointer to the union window

contqueryPtr
pointer to the continuous query object specified in **`dfESPcontquery`**

windowID
user-supplied identifier of the union window

index
primary index. Six types of primary indexes are supported. For more information, see “Understanding Primary and Specialized Indexes” on page 208.

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",
     dfESPindextypes::pi_HASH, true);
```

dfESPpersist

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

Syntax

dfESPpersist *object*(“*directory*”);

Required Arguments

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

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

Examples

Example 1

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

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

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

For more information, see "Persist and Restore Operations" on page 215.

Example 2

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

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

```
// Construct the model but do not start it.
          ....
     // Declare a persist/restore object
     //    "PERSIST" is the directory name where the persisted copy
     //    had been saved. This can be a relative or absolute path
     //
     dfESPpersist persist_restore("PERSIST");
     // Tell the persist/restore object to reload the persisted engine
     //
     bool sucess = persist_restore.restore();
     if (!success) {
```

```
    //
    //    Handle any error -- the engine might not be restored fully
    //    or it could be partially restored, or in a compromised state.
    //
}
// Start all projects from restored state of engine.
dfESPengine::startProjects();
```

Chapter 6

Using Expressions

Overview to Expressions 123
Understanding Data Type Mappings 124
Using Event Metadata in Expressions 124
Using DataFlux Expression Language Global Functions 125
Using Blue Fusion Functions 126

Overview to Expressions

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

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

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

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

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

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

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

expression engine variables used in expression window or window splitter expressions. They can also be useful to declare regular expressions used in expressions.

Initialize expression engines for expression windows that use **dfESPexpression_window::expEngInitializeExp()**. Initialize expression engines for window splitters that use **dfESPwindow::setSplitter()**. You can find examples of both of these in the event stream processing installs in **$DFESP_HOME/examples/cxx**. The window splitter example is named regex, and the expression window example is named splitter_with_initexp.

Understanding Data Type Mappings

An exact data type mapping does not exist between the data types supported by the SAS Event Stream Processing API and those supported by the DataFlux Expression Engine Language.

The following table shows the supported data type mappings.

Table 6.1 *Expression Data Type Mappings Table*

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

Using Event Metadata in Expressions

SAS Event Stream Processing provides a set of reserved words that you can use to access an event's metadata. You can use these reserved words in filter, compute, and join window expressions and in window output splitter expressions. The metadata is not available to pattern window expressions because pattern windows are insert-only.

Reserved Word	Opcode
ESP_OPCODE Use this reserved word to obtain the opcode of an event in a given window.	i — Insert u — Update p — Upsert d — Delete sd — Safe Delete A safe delete does not generate a “key not found” error.
ESP_FLAGS Use this reserved word in expressions to get the flags of an event in a given window.	N — Normal P — Partial R — Retention Delete

For an example, see Appendix 2, “Example: Using a Reserved Word to Obtain an Opcode to Filter Events,” on page 417.

Using DataFlux Expression Language Global Functions

The DataFlux Expression Language supports global functions, also called user-defined functions (UDFs). You can register them as global functions and reference them from any expression window or window splitter expression. For more information about global functions, see *DataFlux Expression Language: Reference Guide*.

There are two SAS Event Stream Processing functions to which you can register global functions:

- **dfESPexpression_window::regWindowExpUDF(*udfString*, *udfName*, *udfRetType*)**
- **dfESPwindow::regSplitterExpUDF(*udfString*, *udfName*, *udfRetType*)**

After you register global functions for a window splitter or an expression window, a splitter expression or a window expression can reference the *udfName*. The *udfName* is replaced with the *udfString* as events are processed.

Filter, compute, join, and pattern expression windows support the use of global functions. Aggregate windows do not support global functions because their output fields are create-only through aggregate functions. All windows support global functions for output splitters on the specified window.

For an example, see Appendix 3, “Example: Using DataFlux Expression Language Global Functions,” on page 421.

Using Blue Fusion Functions

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

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

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

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

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

For an example, see Appendix 4, “Example: Using Blue Fusion Functions,” on page 427.

Chapter 7

Creating Aggregate Windows

Overview to Aggregate Windows **127**

Flow of Operations **128**

Using Aggregate Functions **128**
Overview to Using Aggregate Functions 128
Aggregate Functions for Aggregate Window Field Calculation Expressions 129
Using an Aggregate Function to Add Statistics to an Incoming Event 131
Writing and Using an Aggregate Function 132
Writing Non-Additive Aggregate Functions 133
Writing Additive Aggregate Functions 134

XML Examples of Aggregate Windows **138**

Overview to Aggregate Windows

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

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

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

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

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

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

These groups are internally maintained in the **`dfESPwindow_aggregate`** class as **`dfESPgroupstate`** objects. Each group is collapsed every time that a new event is added or removed from a group by running the specified aggregate functions or

expressions on all non-key fields. The purpose of the aggregate window is to produce one aggregated event per group.

Flow of Operations

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

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

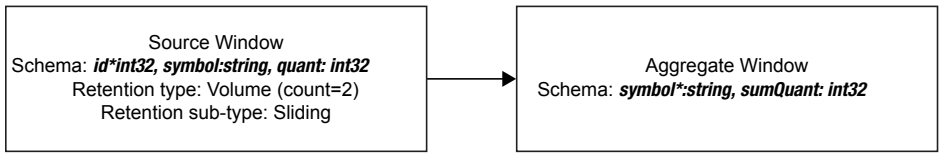
Using Aggregate Functions

Overview to Using Aggregate Functions

During aggregation, events that enter an aggregate window are placed into a group based on the aggregate window's key fields. Aggregate functions are run on each group to compute each non-key field of the aggregate window.

In a sense, the aggregate window partitions input events according to its keys. These partitions are called aggregate groups. The functions that are specified for non-key fields of the aggregate window are special functions. They operate on groups of values and collapse the group to a single scalar value.

Here is a simple example of aggregation:



The key of the aggregate window is **symbol**. It has only one non-key field, **sumQuant**, which is the sum of the field **quant** arriving from the source window.

The function that computes sums of field values is **ESP_aSum(*fieldname*)**. Here, the aggregate window has one non-key field that is computed as **ESP_aSum(quant)**. Conceptually, when an event enters the aggregate window, it is added to the group, and the function **ESP_aSum(quant)** is run, producing a new sum for the group.

Aggregate Functions for Aggregate Window Field Calculation Expressions

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

Aggregate Function	Returns
`ESP_aAve(`*`fieldname`*`)`	average of the group
`ESP_aCount()`	number of events in the group
`ESP_aCountDistinct(`*`fieldname`*`)`	the number of distinct non-null values in the column specified by field name within a group
`ESP_aCountNonNull(`*`fieldname`*`)`	the number of events with non-null values in the column for the specified field name within the group
`ESP_aCountNull(`*`fieldname`*`)`	the number of events with null values in the column for the specified field name within the group
`ESP_aCountOpcodes(`*`opcode`*`)`	count of the number of events matching opcode for group
`ESP_aFirst(`*`fieldname`*`)`	the first event added to the group
`ESP_aGUID()`	a unique identifier
`ESP_aLag(`*`fieldname,lag_value`*`)`	a lag value where the following holds: • **`ESP_aLag(`*`fieldname,`*`0) == ESPaLast(`*`fieldname`*`)`** • **`ESP_aLag(`*`fieldname,`*`1)`** returns the second lag. This is the previous value of *fieldname* that affected the group.
`ESP_aLast(`*`fieldname`*`)`	field from the last record that affected the group
`ESP_aLastOpcodes(`*`opcode`*`)`	the opcode of the last record to affect the group
`ESP_aMax(`*`fieldname`*`)`	maximum of the group
`ESP_aMin(`*`fieldname`*`)`	minimum of the group
`ESP_aMode(`*`fieldname`*`)`	mode, or most popular of the group
`ESP_aStd(`*`fieldname`*`)`	standard deviation of the group

Aggregate Function	Returns
ESP_aSum(*fieldname*)	sum of the group
ESP_aWAve(*weight_fieldname, payload_fieldname*)	weighted group average

The following functions are always additive:

- **ESP_aAve**
- **ESP_aCount**
- **ESP_aCountOpcodes**
- **ESP_aCountDistinct**
- **ESP_aGUID**
- **ESP_aLastOpcode**
- **ESP_aMode**
- **ESP_aStd**
- **ESP_aSum**
- **ESP_aWAve**

The following functions are additive only when they get Inserts:

- **ESP_aCountNonNull**
- **ESP_aCountNull**
- **ESP_aFirst**
- **ESP_aLast**
- **ESP_aMax**
- **ESP_aMin**

You can easily use the built-in aggregate functions for non-key field calculation expressions as follows:

```
dfESPwindow_aggregate *aw_01;
       aw_01 = cq->newWindow_aggregate("aggregateWindow_01",
                                       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 the values determined from the incoming event. They do not need to maintain a group state. This means that if these are the only functions used in a

dfESPwindow_aggrgate instance, special optimizations are made and speed-ups of an order of magnitude in the aggregate window processing can occur.

The **dfESPgroupstate** class is used internally to maintain the groups in an aggregation and an instance of the **dfESPgroupstate** is passed to aggregate functions. The signature of an aggregate function is as follows:

```
typedef dfESPdatavarPtr (*dfESPaggregate_func)(dfESPschema *is,
                                    dfESPeventPtr nep, dfESPeventPtr oep,
                                    dfESPgroupstate *gs);
```

You can find this definition in the **api/dfESPfuncptr.h** file.

The **dfESPgroupstate** object does not only act as a container for a set of events belonging to the same group, but it also maintains a state vector of **dfESPdatavars**, one state vector per non-key field, that can be used by aggregate functions to store a field's state. This enables quick incremental aggregate function updates when a new group member arrives.

Using an Aggregate Function to Add Statistics to an Incoming Event

You can use the **ESP_aLast(fieldName)** aggregate function to pass incoming fields into the aggregate event that is created. This can be useful to add statistics to events through the aggregate window without having to use an aggregate window followed by a join window. Alternatively, using a join window after an aggregate window joins the aggregate calculations or event to the same event that feeds into the aggregate window. But the results in that case might not be optimal.

For example, suppose that this is the incoming event schema:

```
"ID*:int64,symbol:string,time:datetime,price:double"
```

Suppose that with this incoming event schema, you want to add an aggregate statistic:

```
"ID*:int64,symbol:string,time:datetime,price:double,ave_price:double"
```

There, the average is calculated over the group with the same "symbol."

Alternatively, you can define a single aggregate stream, with the following schema:

```
"ID:int64,symbol*:string,time:datetime,price:double,ave_price:double"
```

Note: The group-by is the key of the aggregation, which is "symbol".

Next, use **dfESPwindow_aggregate::addNonKeyFieldCalc(*expression*)** to register the following aggregation functions for each non-key field of this window, which in this case are "ID," "time," "price," and "ave_price":

```
awPtr->addNonKeyFieldCalc("ESP_aLast(ID)");
awPtr->addNonKeyFieldCalc("ESP_aLast(time)");
awPtr->addNonKeyFieldCalc("ESP_aLast(price)");
awPtr->addNonKeyFieldCalc("ESP_aAve(price)");
```

Suppose that the following events come into the aggregate window:

```
insert: 1, "ibm", 09/13/2001T10:48:00, 100.00
insert: 2, "orc", 09/13/2001T10:48:01, 127.00
insert: 3, "ibm", 09/13/2001T10:48:02, 102.00
insert: 4, "orc", 09/13/2001T10:48:03, 125.00
insert: 5, "orc", 09/13/2001T10:48:04, 126.00
```

The aggregate stream produces the following:

```
insert: 1, "ibm", 09/13/2001T10:48:00, 100.00, 100.00
insert: 2, "orc", 09/13/2001T10:48:01, 127.00, 127.00
update: 3, "ibm", 09/13/2001T10:48:00, 102.00, 101.00
update: 4, "orc", 09/13/2001T10:48:01, 125.00, 126.00
update: 5, "orc", 09/13/2001T10:48:01, 126.00, 126.00
```

By using **aLast(*fieldname*)** and then adding the aggregate fields of interest, you can avoid the subsequent join window. This makes the modeling cleaner.

Writing and Using an Aggregate Function

Write aggregate functions with zero, one, two or three arguments. The arguments must be either integer-valued DataFlux expressions, integer constants, or symbols in the input schema for the aggregation.

The most commonly used aggregate functions are one parameter functions with an input schema symbol (for example, the built-in aggregation function **ESP_aMax(*fieldname*)**). For field names in the input schema, the field index into the input event is passed into the aggregate function, not the value of the field. This is important when you deal with groups. You might need to iterate over all events in the group and extract the values by event index from each input event.

After you write an aggregate function, embed it in C++ code in order to use it in your event stream processing application. How to do this is documented in an example provided in **$DFESP_HOME/examples/cxx/aggregate_userdef**.

Copy your function into **$DFESP_HOME/examples/cxx/aggregate_userdef/src/functions.cpp**. Suppose your function is named **My_Aggregation_Function**. At the bottom of **functions.cpp**, create a wrapper function for your aggregation function.

```
// the uMyFunction wrapper:
     // every aggregation function must be wrapped like this.
    //
    int dfESPaggrfunc_uMyFunctionWrapper(dfESP_EXPengine_t *e,
                                         dfESP_EXPsym_value_t *returnval,
                                         int parmcount,
                                         dfESP_EXPsym_value_t **parms)
                                          {
        return dfESPaggrfunc_Wrapper((void *)my_aggrergation_function, e,
                                            returnval, parmcount, parms);
                                          }
```

Create an entry in the user-defined function list for the wrapper function.

```
// SAS Event Stream Processing calls this function to get all user-defined
    //    aggregation functions during the initialization stage.
    //
    void add_user_aggrFunctions() {
        dfESPengine *e = dfESPengine::getEngine();

        dfESPptrList<aggr_function_t *> &uFuncts = e->getUDAFs();

        // push back as many user defined functions as you like:
        // the parameters are: <callable name>, <function pointer>,
        // <num args>, <additive flag>, <additive flag for insert only>,
        // <description>
```

```
    uFuncts.push_back(new aggr_function_t("USER_myFunction",
        (void *)dfESPaggrfunc_uMyFunctionWrapper,
                1, true, true, "description"));
}
```

Adjust the number of arguments, additive flags, and the description field accordingly. The sample code **`$DFESP_HOME/examples/cxx/aggregate_userdef/src/functions.cpp`** provides two complete examples. The makefile distributed with the sample code produces a shared library (**`libdfxesp_udafD-`*`major.minor`***).

Writing Non-Additive Aggregate Functions

The simplest aggregate sum function does not maintain state and is not additive. The function iterates through each event in a group to aggregate. It requires the aggregation window to maintain a copy of every input event for all groups.

The following code performs these basic steps:

1. Look at the input and output types and verify compatibility.
2. Initialize a return variable of the specified output type.
3. Loop across all events in the group and perform the aggregation function.
4. Check for computational errors and return the error or the result .

```
// a non-additive summation function
    //
    // vgs is the groupstate object passed as a (void *) pointer
    // fID is the filed ID in internal field order of the field on
    //    which we sum.
    dfESPdatavarPtr uSum_nadd(void *vgs, size_t fID) {

        dfESPdatavar    *rdv;
        // placeholder for return value
        dfESPgroupstate *gs = (dfESPgroupstate *)vgs;
        // the passed groupstate cast back to dfESPgroupstate object.

        // get the 1) aggregate schema (output schema)
        //     and 2) the schema of input events
        //
        dfESPschema  *aSchema = gs->getAggregateSchema();
        dfESPschema  *iSchema = gs->getInputSchema();

        // get the type of 1) the field we are computing in the aggregate schema
        // and 2) the input field we are summing.
        //
        dfESPdatavar::dfESPdatatype aType =
          aSchema->getTypeEO(gs->getOperField());
        dfESPdatavar::dfESPdatatype iType =
          iSchema->getTypeIO(fID);

        dvn_error_t   retCode = dvn_noError;
        // return code for using the datavar numerics package.

        // If the input fields or the output field is non-numeric,
        // flag an error.
        //
```

```
if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
cerr << "summation must work on numeric input, produce numeric output."
     << endl;
return NULL;
}

// in the ESP type system, INT32 < INT64 < DOUBLE < DECSECT.
//   This checks compatibility. The output type must be greater
//   equal the input type. i.e. we cannot sum a column of int64
//   and puit them into an int32 variable.
//
if (iType > aType) {
 cerr << "output type is not precise enough for input type" << endl;
 return NULL;
}

dfESPeventPtr nev = gs->getNewEvent();
dfESPeventPtr oev = gs->getOldEvent();

// create the datavar to return, of the output type and set to zero.
//
rdv = new dfESPdatavar(aType);     // NULL by default.
rdv->makeZero();


dfESPeventPtr gEv = gs->getFirst(); // get the first event in the group.
dfESPdatavar iNdv(iType);           // a place to hold the input variable.
while (gEv) {                       // iterarate until no more events.
    gEv->copyByIntID(fID, &iNdv);   // extract value from record into iNdv;

    if (!iNdv.isNull()) {           // input not null
        if ((retCode = dv_add(rdv, rdv, &iNdv)) != dvn_noError)
            break;                  // rdv = add(rdv, iNdv)
    }
    gEv = gs->getNext();            // get the first event in the group.
}

if (retCode != dvn_noError ) {       // if any of our arithmitic fails.
 rdv->null();                        // return a null value.
 cerr << "uSum() got an arithmetic error in summing up values" << endl;
}

return rdv;
```

Writing Additive Aggregate Functions

Aggregate functions that compute themselves based on previous field state and a new field value are called additive aggregation functions. These functions provide computational advantages over aggregate functions.

An additive aggregate function can be complex for two reasons:

- They must look at the current state (for example, the last computed state).
- They must evaluate the type of incoming event to make proper adjustments.

Suppose that you keep the state of the last value of the group's summation of a field. When a new event arrives, you can conditionally adjust the state base on whether the incoming event is an Insert, Delete, or Update. For an Insert event, you simply increase the state by the new value. For a Delete, you decrease the state by the deleted value. For an Update, you increase and decrease by the new and old values respectively. Now the function never has to loop through all group values. It can determine the new sum based on the previous state and the latest event that affects the group.

The following code performs these basic steps:

1. Look at the input and output types and verify compatibility.
2. Initialize a return variable of the specified output type.
3. Determine whether the function has been called before. That is, is there a previous state value?
   - If so, retrieve it for use.
   - If not, create a new group with an arriving insert so that you can set the state to the incoming value.
4. Switch on the opcode and adjust the state value.
5. Check for computational errors and return the error value or the state value as the result.

```
// an additive summation function
    //
    // vgs is the groupstate object passed as a (void *) pointer
    // fID is the filed ID in internal field order of the field on
    //    which we sum.
    dfESPdatavarPtr uSum_add(void *vgs, size_t fID) {

        dfESPdatavar    *rdv;
        // placeholder for return value
        dfESPgroupstate *gs = (dfESPgroupstate *)vgs;
        // the passed groupstate cast back to dfESPgroupstate object.

        // get the 1) aggregate schema (output schema)
        // and 2) the schema of input events
        //
        dfESPschema  *aSchema = gs->getAggregateSchema();
        dfESPschema  *iSchema = gs->getInputSchema();

        // get the type of 1) the field we are computing in the aggregate schema
        // and 2) the input field we are summing.
        //
        dfESPdatavar::dfESPdatatype aType =
          aSchema->getTypeEO(gs->getOperField());
        dfESPdatavar::dfESPdatatype iType =
          iSchema->getTypeIO(fID);


        dvn_error_t   retCode = dvn_noError;
        // return code for using the datavar numerics package.

        // If the input fields or the output field is non-numeric,
        // flag an error.
        //
```

```
if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
cerr << "summation must work on numeric input, produce numeric output."
     << endl;
return NULL;
}

// in the ESP type system, INT32 < INT64 < DOUBLE < DECSECT.
//   This checks compatibility. The output type must be greater
//   equal the input type. i.e. we cannot sum a column of int64
//   and puit them into an int32 variable.
//
if (iType > aType) {
cerr << "output type is not precise enough for input type" << endl;
return NULL;
}

// fetch the input event from the groupstate object (nev)
// and, in the case of an update, the old event that
// is being updated (oev)
//
dfESPeventPtr nev = gs->getNewEvent();
dfESPeventPtr oev = gs->getOldEvent();

// Get the new value out of the input record
//
dfESPdatavar iNdv(iType);
// a place to hold the input variable.
dfESPdatavar iOdv(iType);
// a place to hold the input variable (old in upd case).
nev->copyByIntID(fID, iNdv);
// extract input value (no copy) to it (from new record)

// Get the old value out of the input record (update)
//
if (oev) {
    oev->copyByIntID(fID, iOdv);
// extract input value to it (old record)
}

// Note: getStateVector() returns a reference to the state vector for
//       the field we are computing inside the group state object.
//
dfESPptrVect<dfESPdatavarPtr> &state = gs->getStateVector();

// create the datavar to return, of the output type and set to zero.
//
rdv = new dfESPdatavar(aType);     // NULL by default.
rdv->makeZero();

// If the state has never been set, we set it and return.
//
if (state.empty()) {
    dv_assign(rdv, &iNdv);
    //  result = input
    state.push_back(new dfESPdatavar(rdv));
    //  make a copy and push as state
```

```
        return rdv;
    }

    // at this point we have a state,
    // so lets see how we should adjust it based on opcode.
    //

    dfESPeventcodes::dfESPeventopcodes opCode = nev->getOpcode();
    bool  badOpcode = false;
    int   c = 0;
    switch (opCode)  {
    case dfESPeventcodes::eo_INSERT:
        if (!iNdv.isNull())
            retCode = dv_add(state[0], state[0], &iNdv);
        break;
    case dfESPeventcodes::eo_DELETE:
        if (!iNdv.isNull())
            retCode = dv_subtract(state[0], state[0], &iNdv);
        break;
    case dfESPeventcodes::eo_UPDATEBLOCK:
        retCode = dv_compare(c, &iNdv, &iOdv);
        if (retCode != dvn_noError) break;
        if (c == 0)  // the field value did not change.
            break;
        if (!iNdv.isNull())
        // add in the update value
            retCode = dv_add(state[0], state[0], &iNdv);
        if (retCode != dvn_noError) break;
        if (!iOdv.isNull())
        // subtract out the old value
            retCode = dv_subtract(state[0], state[0], &iOdv);
        break;
    default:
    cerr << "got a bad opcode when running uSum_add()" << endl;
    badOpcode = true;
    }

    if ( badOpcode || (retCode != dvn_noError) ) {
        rdv->null();
        // return a null value.
    cerr << "uSum() got an arithmetic error summing up values" << endl;
    } else
        dv_assign(rdv, state[0]);
    // return the adjusted state value

    return rdv;
}
```

Even though it is possible to use the **`Sum()`** aggregate function to iterate over the group and compute a new sum when a new group changes, faster results are obtained when you maintain the **`Sum()`** in a **`dfESPdatavar`** in the **`dfESPgroupstate`** object and increment or decrement the object by the incoming value, provided that the new event is an Insert, Update, or Delete. The function then adjusts this field state so that it is up-to-date and can be used again when another change to the group occurs.

XML Examples of Aggregate Windows

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

```
<window-aggregate name='aw_01'>
  <schema>
    <fields>
       <field name='symbol' type='string' key='true'/>
       <field name='totalQuant' type='int32'/>
       <field name='maxPrice' type='double'/>
    </fields>
   </schema>
   <output>
     <field-plug function='summationAggr' plugin='libmethod' additive='true'/>
     <field-plug function='maximumAggr' plugin='libmethod' additive='false'/>
   </output>
   <connectors>
      <connector class='fs'>
         <properties>
            <property name='type'>sub</property>
            <property name='fstype'>csv</property>
            <property name='fsname'>aggregate.csv</property>
            <property name='snapshot'>true</property>
         </properties>
      </connector>
    </connectors>
```

```
</window-aggregate>
```

Chapter 8

Creating Counter Windows

Overview to Counter Windows **141**

Examples **142**
XML 142
C++ 143

Overview to Counter Windows

A counter window enables you to see how many events are going through your model and the rate at which they are being processed.

The XML element to create a counter window is as follows:

```
<window-counter name='name' count-interval='period' clear-interval='cperiod'/>
```

Field	Description
`name`	The name of the window.
`count-interval` (optional)	Period of time for which to generate a current event processing rate, specified in **`'[value] [unit]'`**.
`clear-interval` (optional)	Period of inactivity after which all counter data is reset to 0, specified in **`'[value] [unit]'`**.

When you specify a **`count-interval`**, the counter window reports performance statistics regularly at that interval. Event generation can be driven by either the arrival of an event or by the window receiving a heartbeat. The window checks to see whether it is time to report the values and generate an event. This event contains overall values plus the interval values:

```
<window-counter name='counter'
                      count-interval='2 seconds'
                      clear-interval='30 seconds'/>
   <event opcode='upsert' window='trades/trades/counter'>
      <value name='input'>trades</value>
      <value name='intervalCount'>288215</value>
      <value name='intervalRate'>144108</value>
      <value name='intervalSeconds'>2</value>
```

```
      <value name='totalCount'>794312</value>
      <value name='totalRate'>132385</value>
      <value name='totalSeconds'>6</value>
   </event>
```

If you do not specify **`count-interval`**, an event with performance numbers is generated each time the window receives a heartbeat. This event contains only overall values:

```
<window-counter name='counter'/>
   <event opcode='upsert' window='trades/trades/counter'>
      <value name='input'>trades</value>
      <value name='totalCount'>7815189</value>
      <value name='totalRate'>132461</value>
      <value name='totalSeconds'>59</value>
   </event>
```

Examples

XML

```
<window-counter name='counter' count-interval='2 seconds'
                       clear-interval='30 seconds'/>
...
<edges>
   <edge source='source' target='largeTrades counter'/>
   ...
</edges>
...
<event opcode='upsert' window='project/query/counter'>
   <value name='input'>source</value>
   <value name='intervalCount'>205418</value>
   <value name='intervalRate'>102709</value>
   <value name='intervalSeconds'>2</value>
   <value name='totalCount'>205418</value>
   <value name='totalRate'>102709</value>
   <value name='totalSeconds'>2</value>
</event>

<event opcode='upsert' window='project/query/counter'>
   <value name='input'>source</value>
   <value name='intervalCount'>293771</value>
   <value name='intervalRate'>146886</value>
   <value name='intervalSeconds'>2</value>
   <value name='totalCount'>499189</value>
   <value name='totalRate'>124797</value>
   <value name='totalSeconds'>4</value>
</event>

<event opcode='upsert' window='project/query/counter'>
   <value name='input'>source</value>
   <value name='intervalCount'>283473</value>
   <value name='intervalRate'>141736</value>
   <value name='intervalSeconds'>2</value>
```

```
   <value name='totalCount'>782662</value>
   <value name='totalRate'>130444</value>
   <value name='totalSeconds'>6</value>
</event>

<event opcode='upsert' window='project/query/counter'>
   <value name='input'>source</value>
   <value name='intervalCount'>217338</value>
   <value name='intervalRate'>108669</value>
   <value name='intervalSeconds'>2</value>
   <value name='totalCount'>1000000</value>
   <value name='totalRate'>125000</value>
   <value name='totalSeconds'>8</value>
</event>
```

C++

```
dfESPproject        *project = engine->newProject("project");
dfESPcontquery        *contquery = project->newContquery("contquery");
dfESPwindow_source    *source = contquery->newWindow_source("source",
                              dfESPindextypes::pi_RBTREE,
                              dfESPstring("ID*:int32,symbol:string,valstr:string"));
dfESPschema        *schema = source->getSchema();

dfESPwindow_counter    *counter = contquery->
                       newWindow_counter("counterWindow",dfESPindextypes::pi_RBTREE);
counter->setCountInterval("2 seconds");
counter->setClearInterval("10 seconds");

contquery->addEdge(source,0,counter);
```

Chapter 9

Creating Functional Windows

Overview to Functional Windows **145**

Using Event Loops **145**

Understanding and Using Function Context **146**
Overview to Function Context 146
Types of Functions You Can Use 146
Using Expressions 147
Specifying Properties 147
Function-Context Example 150

Functional Window Examples **151**
Stock Trades 151
C++ Code 152

Overview to Functional Windows

A functional window enables you to use different types of functions to manipulate or transform event data. You define a schema and then a *function context* that contains functions and supporting entities such as regular expressions, XML, and JSON. When an event enters a functional window, the window looks for a function with a name that corresponds to each field in its schema. If the function exists, it is run and the resulting value is entered into the output event. If no function is specified for a field, and a field with the same name exists in the input schema, the input value is copied directly into the output event.

You can use the XML element **`<generate>`** to specify a function to run in order to determine whether you want to generate an event from an input event.

You can generate multiple output events from a single input event using the **`<event-loop>`** element. You can specify a function to create some type of data and then grab any number of entities from that created data. For each of these entities, you can generate an event using a function context specific to that event loop.

Using Event Loops

Event loops enable you to generate any number of events from a single input event. You can specify any number of event loops. Each loop deals with a particular type of data.

For each input event, a functional window does the following for each event loop entry:

1. Uses a function or reference to generate the data to be used as input to the loop. For example, in an **event-loop-xml** loop, you would specify the **use-xml** element to generate valid XML. This content can be either a function or a reference to a property in the window's function-context.
2. Applies an appropriate expression, such as XPATH or JSON, to the data to retrieve 0 or more entities.
3. For each of these entities, sets a data item specified by the data attribute to the string value of the entity. Then, any functions in the function-context are run and an event is generated. Any property or event value in the window's function context is accessible to the loop's function context. Also, the variable specified by the data attribute is accessible via **'$'** notation.

Understanding and Using Function Context

Overview to Function Context

The function context enables you to define functions within a functional window. You can use regular expressions, XML and XPATH, JSON, and other capabilities to transform data from different types of complex input into usable output.

Types of Functions You Can Use

You can use two types of functions within a function context:

- general functions
- functions specific to event stream processing

You can reference event fields in either the input event or the output event using the '$' notation: **$[*name of field*]**

Here are the data mappings relevant to using functions in a function context:

string	ESP_UTF8STR
float	ESP_DOUBLE
long	ESP_INT64
integer	ESP_INT32
Boolean	ESP_INT32

For example, suppose that you have a **name** field in the input event and you wanted to generate an **occupation** field in the output event. You could code the function as follows:

```
<function name='occupation'>
ifNext
```

```
(
   equals($name,'larry'),'plumber',
   equals($name,'moe'),'electrician',
   equals($name,'curly'),'carpenter'
)
</function>
```

You can also reference fields in the output event. Continuing the above example, perhaps you want to add the **hourlyWage** field to the output event depending on the value of **occupation**:

```
<function name='hourlyWage'>
ifNext
(
   equals($occupation,'plumber'),85.0,
   equals($occupation','electrician'),110.0,
   equals($occupation,'carpenter'),60.0
)
</function>
```

Note: It is critical to pay attention to the sequence of fields when you define functions. If a function references an output event field, then that field must be computed before the referring field.

Using Expressions

Use the **<expressions>** element to specify POSIX regular expressions that are compiled a single time.

```
<expressions>
      <expression name='expname'>[posix_regular_expression]</expression>
      ...
   </expressions>
```

After you specify an expression, you can reference it from within a function using the following notation:

```
<function name='myData'>rgx(#expname,$inputField,1)</function>
```

Suppose that you were getting a data field that contained a URI, and you wanted to extract the protocol from the URI. If you use **<function name='protocol'>rgx('(.*):',$uri,1)</function>**, the regular expression is compiled each time that the function is run. However, if you use the following code:

```
<expressions>
   <expression name='getProtocol'>(.*):</expression>
</expressions>

<function name='protocol'>rgx(#getProtocol,$uri,1)</function>
```

The expression is compiled a single time and used each time that the function is run.

Specifying Properties

Properties are similar to expressions in that they are referenced from within functions using the '#' notation: **#[property-*type*]**

There are five types of properties:

- **map** executes the function to generate a map of name-value pairs to be used for value lookups by name
- **set** executes the function to generate a set of strings to be used for value lookups
- **XML** executes the function to generate an XML object that can be used for XPATH queries.
- **JSON** executes the function to generate a JSON object that can be used for JSON lookups
- **string** executes the function to generate a string for general use in functions

Each property is generated using functions. These functions can reference properties defined before them in the XML.

Here is how you would code each property type

Property Type	Description
<property-map name='*name*' outer='*outdelim*' inner='*indelim*'>[*code*]</property-map>	• **name** - the name of the property • **outer** - the outer delimiter to use in parsing the data • **inner** - the inner delimiter to use in parsing the data • ***code*** - the function to run to generate the data to be parsed into a name-value map For example, suppose there exists an input field **data** that looks like this: **firstname=joe;lastname=smith;occupation=software** You could create the following property-map: **<property-map name='myMap' outer=';' inner='='>$data</property-map>**
<property-xml name='*name*'>[*code*]</property-xml>	• **name** - the name of the property • ***code*** - the function to run to generate valid XML
<property-json name='*name*'>[*code*]</property-json>	• **name** - the name of the property • ***code*** - the function to run to generate valid JSON
<property-string name='*name*'>[*code*]</property-string>	• **name** - the name of the property • ***code*** - the function to run to generate a string value

Property Type	Description
<property-set name='*name*' delimiter='*delim*'>[*code*]</ property-set>	• **name** - the name of the property • **delimiter** - the delimiter to use in parsing the data • ***code*** - the function to run to generate the data to be parsed into a value set For example, suppose there exists an input field data that looks like this: **ibm,sas,oracle** This would yield the following property set: **<property-set name='mySet' delimiter=','>$data</ property-set>**

Suppose you had some employee information streaming into the model.

```
<event>
   <value name='map'>name:[employee name];
          position:[employee position]
   </value>
   <value name='developerInfo'>
       <![CDATA[<info>this is developer info</info>]]>
   </value>
   <value name='managerInfo'>
       <![CDATA[<info>this is manager info</info>]]>
   </value>
</event>
```

You can create a **property-map** to store employee data and then examine the position field in order to create a **property-xml** containing the appropriate data. If the employee is a developer, the XML is from **developerInfo**. Otherwise, it uses **managerInfo**. Your function-context would look like this:

```
<function-context>
   <properties>
      <property-map name='myMap' outer=';' inner=':'>$map</property-map>
      <property-xml name='myXml'>if(equals(mapValue(#myMap,'position'),'developer'),
       $developerInfo,$managerInfo)
      </property-xml>
   </properties>
   <functions>
      <function name='employee'>mapValue(#myMap,'name')</function>
      <function name='info'>xpath(#myXml,'text()')</function>
   </functions>
</function-context>
```

Streaming in the following event:

```
<event>
   <value name='map'>
     name:curly;position:developer<
   /value>
   <value name='developerInfo'>
     <![CDATA[<info>this is developer info</info>]]>
   </value>
```

```
   <value name='managerInfo'>
     <![CDATA[<info>this is manager info</info>]]>
   </value>
</event>

<event>
   <value name='map'>name:moe;position:manager</value>
   <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
   <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>
```

Yields the following result:

```
<event opcode='insert' window='project/query/transform'>
   <value name='employee'>curly</value>
   <value name='id'>fd26bf36-3d65-4d17-8dc6-317409bbf5b6</value>
   <value name='info'>this is developer info</value>
</event>

<event opcode='insert' window='project/query/transform'>
   <value name='employee'>moe</value>
   <value name='id'>84c56bb7-9f3c-4cb8-93a5-8dc2f75d353b</value>
   <value name='info'>this is manager info</value>
</event>
```

Function-Context Example

```
<function-context>
   <expressions>
      <expression name='myexp'>posix_regular_expression</expression>
      ...
   </expressions>
   <properties>
      <property-map name='map' outer=';' inner='='>code</property-map>
      <property-xml name='xmlprop'>code</property-xml>
      <property-json name='jsonprop'>code</property-json>
      <property-string name='string'>code</property-string>
      <property-set name='mySet' delimiter=','>code</property-set>
      ...
   </properties>
   <functions>
      <function name='mySum'>code</function>
      ...
   </functions>
</function-context>
```

Functional Window Examples

Stock Trades

Suppose that you had stock trade information streaming into an event stream processing model. You want to generate an event anytime that a huge trade (> 150000 shares) takes place during the first or last 15 minutes of the trading day.

Your model includes the following source window:

```
<window-source name='source' insert-only='true'>
   <schema>
      <fields>
         <field name='id' type='int32' key='true'/>
         <field name='symbol' type='string'/>
         <field name='currency' type='int32'/>
         <field name='time' type='int64'/>
         <field name='msecs' type='int32'/>
         <field name='price' type='double'/>
         <field name='quant' type='int32'/>
         <field name='venue' type='int32'/>
         <field name='broker' type='int32'/>
         <field name='buyer' type='int32'/>
         <field name='seller' type='int32'/>
         <field name='buysellflg' type='int32'/>
      </fields>
   </schema>
</window-source>
```

You stream events from the source window into a filter window to obtain huge trades:

```
<window-filter name='hugeTrades'>
   <expression><![CDATA[quant>150000]]></expression>
</window-filter>
```

That data flows into the functional window:

```
<window-functional name='transform'>
   <schema>
      <fields>
         <field name='id' type='int32' key='true'/>
         <field name='symbol' type='string'/>
         <field name='timeString' type='string'/>
         <field name='hourOfDay' type='double'/>
         <field name='quant' type='int32'/>
      </fields>
   </schema>
   <function-context>
      <functions>
         <function name='timeString'>
            timeString($time)
         </function>
         <function name='hourOfDay'>
            precision(quotient(timeSecondOfDay($time),3600),2)
          </function>
```

```
      </functions>
   </function-context>
</window-functional>
```

Several fields in are defined in the schema of the functional window. However, functions are defined only for a few fields. The remaining fields are copied from the input event.

You include functions to do the following things: format a time into a readable form in the **timeString** field, and then calculate a floating-point value representing the hour of the day on which the trade was made.

```
<function name='hourOfDay'>
         precision(quotient(timeSecondOfDay($time),3600),2)
       </function>
```

This function produces a floating-point value that you can stream to a filter in order to get early and late huge trades.

```
<window-filter name='earlyTrade'>
   <expression><![CDATA[hourOfDay<9.75]]></expression>
</window-filter>

<window-filter name='lateTrade'>
   <expression><![CDATA[hourOfDay>15.75]]></expression>
</window-filter>
```

Running several million trades through the model generates output like the following:

```
<event opcode='insert' window='project/query/earlyTrade'>
   <value name='hourOfDay'>9.73</value>
   <value name='id'>11847604</value>
   <value name='quant'>1000000</value>
   <value name='symbol'>NOK</value>
   <value name='timeString'>Wed Aug  4 09:43:48 2010</value>
</event>

<event opcode='insert' window='project/query/lateTrade'>
   <value name='hourOfDay'>15.81</value>
   <value name='id'>16739499</value>
   <value name='quant'>270400</value>
   <value name='symbol'>TXT</value>
   <value name='timeString'>Wed Aug  4 15:48:47 2010</value>
</event>
```

C++ Code

```
dfESPwindow_functional    *transform =
       contquery->newWindow_functional("transform",dfESPindextypes::pi_EMPTY,
                                        "id*:int64,symbol:string,timeString:string,
                                        hourOfDay:double,quant:int64");
transform->setInsertOnly();
transform->getContext()->setFunction("timeString","timeString($time)");
transform->getContext()->setFunction("hourOfDay",
                            "precision(quotient(timeSecondOfDay($time),3600),2)");
contquery->addEdge(hugeTrades,0,transform);
```

Chapter 10

Creating Join Windows

Overview to Join Windows **153**

Understanding Streaming Joins **154**
Overview to Streaming Joins 154
Using Secondary Indices 156
Using Regeneration versus No Regeneration 156

Creating Empty Index Joins **157**

Examples of Join Windows **157**

Overview to Join Windows

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

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

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

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

A join window produces a single output stream of joined events. Because an 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.

Understanding Streaming Joins

Overview to Streaming Joins

Given a left window, a right window, and a set of join constraints, a streaming join can be classified into one of three different types.

Join Type	Description
one-to-one joins	An event on either side of the join can match at most one event from the other side of the join. This type of join always involves two dimension tables.
one-to-many joins (or many-to-one joins)	An event that arrives on one side of the join can match many rows of the join. An event that arrives on the other side of the join can match at most one row of the join. There are two conditions for a join to be classified as a one-to-many (or many-to-one) join: • a change to one side of the join can affect many rows • a change to the other side can affect at most one row Both conditions must be met.
many-to-many joins	A single event that arrives on either side of the join can match more than one event on the other side of the join.

The join constraints are an n-tuple of equality expressions. Each expression involves one field from the left window and one field from the right. For example: $(left.f_1 == right.f_{10}), (left.f_2 == right.f_7), \ldots (left.field_{10} == right.field_1)$.

In a streaming context, every window has a primary key that enables the insertion, deletion, and updating of events. The keys for a join window are derived from the total set of keys from the left window and the right window. When an event arrives on either side, you must be able to compute how the join changes, given the nature of the arriving data (Insert, Update, or Delete). The theory of join-key derivation that SAS Event Stream Processing follows maintains consistency for the most common join cases.

Some of the basic axioms used in the join-key derivation are as follows:

- For a left-outer join, the keys of the join cannot contain any keys from the right window. A joined event is output when a left event arrives. There is no matching event on the right.
- For a right-outer join, the keys of the join cannot contain any keys from the left window. A joined event is output when a right event arrives. There is no matching event on the left.
- For a many-to-many join, the keys of the joins need to be the union of the keys for the left and right windows. To understand this axiom, think of an event coming in on one side of the join that matches multiple events on the other side of the join. In this case, all the keys of the many side must be included. Otherwise, you cannot distinguish the produced events. Because the single event that matches many events can come on the other side of the join, reverse the above statement to determine what

happens in a streaming context. All keys from the left and right side of the join must be present.

- For one-to-many or many-to-one joins, the side of the join that matches multiple events is the side from which that the join windows keys derive. This is the case when there is a single event on the other side.

Join windows are either dimension windows or fact windows. Dimension windows are those whose entire set of key fields participate in the join constraints. Fact windows are those that have at least one key field that does not participate in the join constraints.

The following table summarizes the allowed join sub-types and key derivation based on the axioms and the specified join-constraints.

Classification	Left Window	Right Window	Allowed Type	Key Derivation
one-to-one	Dimension	Dimension	Left Outer	join keys are keys of left window
			Right Outer	join keys are keys of right window
			Full Outer	join keys are keys of left window (arbitrary choice)
			Inner	join keys are keys of left window (arbitrary choice)
many-to-one	Fact	Dimension	Left Outer	join keys are keys of left window (right window is lookup)
			Inner	join keys are keys of left window (right window is lookup)
one-to-many	Dimension	Fact	Right Outer	join keys are keys of right window (left window is lookup)
			Inner	join keys are keys of right window (left window is lookup)
many-to-many	Fact	Fact	Inner	join keys are the full set of keys from the left and right windows

Note: When all the keys of a window are used in a join constraint, adding additional non-key fields on the side of the constraint is not honored by SAS Event Stream Processing. For example, suppose that the set of left-hand fields that participate in a join constraint contain all of the keys of the left window. Any non-key fields in that set are ignored.

Using Secondary Indices

For allowed one-to-many and many-to-one joins, a change to the fact table enables immediate lookup of the matching record in the dimension table through its primary index. All key values of the dimension table are mapped in the join constraints. 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. By default, matching records in the fact table are sought through a table scan.

For very limited changes to the dimension table there is no additional secondary index maintenance, so the join processing can be optimized. Here, the dimension table is a static lookup table that can be pre-loaded. All subsequent changes happen on the fact table.

When a large number of changes are possible to the dimension table, it is suggested to enable a secondary index on the join. Automatic secondary index generation is enabled by specifying a join parameter when you construct a new join window. This causes a secondary index to be generated and maintained automatically when the join type involves a dimension table. This has the advantage of eliminating all table scans when changes are made to the dimension table. There is a slight performance penalty when you run with secondary indices turned on. 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, you can achieve time savings of two to three orders of magnitude through the use of secondary indices.

For many-to-many joins, enabling on secondary indices is recommended.

Using Regeneration versus No Regeneration

The default join behavior is to always regenerate the appropriate rows of a join window when a change is made to either side of the joins. The classic example of this is a left outer join: the right window is the lookup window, and the left table is the fact (streaming) window. The lookup side of the join is usually pre-populated, and as events stream though the left window, they are matched and the joined events output. Typically, this is a one-to-one relation for the streaming side of the join: one event in, one combined event out. Sometimes a change is made on the dimension side. This change can be in the form of an update to an event, a deletion of an event, or an insertion of a new event. The default behavior is to issue a change set of events that keeps the join consistent.

In regeneration mode, the behavior of a left outer join on a change to the right window (lookup side) is as follows:

- Insert: find all existing fact events that match the new event. If any are found, issue an update for each of these events. They would have used nulls for fields of the lookup side when they were previously processed
- Delete: find fact events that match the event to be deleted. If any are found, issue an update for each of these events. They would have used matching field values for the lookup event, and now they need to use nulls as the lookup event is removed.
- Update: Behaves like a delete of the old event followed by an insert of the new event. Any of the non-key fields of the lookup side that map to keys of the streaming side are taken into account. It is determined whether any of these fields changed value.

With no-regeneration mode, when there is a left outer join on a change to the right window (lookup side), changes to the dimension (lookup) table affect only new fact

events. All previous fact events that have been processed by the join are not regenerated. This frequently occurs when a new dimension window is periodically flushed and re-loaded.

The join window has a **no-regenerates** flag that is false by default. This gives the join full-relational join semantics. Setting this flag to true for your join window enables the **no-regenerates** semantics. Setting the flag to true is permitted for any of the one-to-many (or many-to-one) joins and the one-to-one outer joins. When a join window is running in **no-regenerates** mode, it optimizes memory usage by omitting the reference-counted copy of the fact window's index that is normally maintained in the join window.

Creating Empty Index Joins

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

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

Examples of Join Windows

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

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

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

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

```
dfESPwindow_join *jw;
jw = cq->newWindow_join("myJoinWindow", dfESPwindow_join::jt_LEFTOUTER,
                        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,
                        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.

Chapter 11

Creating Notification Windows

Overview to Notification Windows **159**

Notification Window Delivery Channels **161**
Overview to Notification Window Delivery Channels 161
Using the Email Delivery Channel 161
Using the SMS Delivery Channel 162
Using the MMS Delivery Channel 163

Using the Function-Context Element **164**

Examples of Notification Windows **167**
Building a Streaming Performance Monitor 167
Catching Front Running Traders 168
Example Written in C++ 173

Overview to Notification Windows

Notification windows enable you to send notifications through email (SMTP), text (SMS), and or multimedia message (MMS). These windows, like functional windows, enable you to define a function context to transform incoming events before processing them for possible notifications. Each of the different types of notification has its own configuration requirements. For example, an email requires that the configuration specify the event field that contains the 'send to' email address. SMS and MMS require phone numbers and phone provider gateway information.

You can format notifications as you want and include the event values within the message. To include event values, include the name of the field, preceded by a **$** character, in your message formatting:

```
<b>$broker</b> sold $quant1 shares of <b>$symbol</b>
$tstap1 for self for $$price1, then sold $quant2 shares for customer at $tstamp2.
```

Notification windows enable you to create any number of delivery channels to send notifications. You can specify functions to determine whether to send the notification. Given the potentially massive amounts of streaming data that could cause an avalanche of notifications, you can specify a throttle interval for each channel. If you set the interval to '1 hour', you send at most one notification from that channel to any recipient every hour.

Notification windows never generate events. Nevertheless, you can use the **schema** element to specify values for the function-context to generate. You can use these values to format notification messages.

The full XML configuration of a notification window is as follows:

```
<window-notification name=''>
   <schema>
   ...
   </schema>

   <function-context>...</function-context>

   <smtp host='host'
         user='user'
         password='password'
         port='port' (opt, default='25') />

   <delivery-channels>

      <email throttle-interval='' test='true | false'>
         <deliver>[code]</deliver>
         <email-info>
            <sender>[code]</sender>
            <recipients>[code]</recipients>
            <subject>[code]</subject>
            <from>[code]</from>
            <to>[code]</to>
         </email-info>
         <email-contents>
            <text-content name=''>...</text-content>
            <html-content name=''>...</html-content>
            <image-content name=''>...</image-content>
            ...
         </email-contents>
      </email>

      <sms throttle-interval='' test='true | false'>
         <deliver>[code]</deliver>
         <sms-info>
            <sender>[code]</sender>
            <subject>[code]</subject>
            <from>[code]</from>
            <gateway>[code]</gateway>
            <phone>[code]</phone>
         </sms-info>
         <sms-contents>
            <text-content name=''>...</text-content>
         </sms-contents>
      </sms>

      <mms throttle-interval='' test='true | false'>
         <deliver>[code]</deliver>
         <mms-info>
            <sender>[code]</sender>
            <subject>[code]</subject>
            <gateway>[code]</gateway>
```

```
            <phone>[code]</phone>
         </mms-info>
         <mms-contents>
            <text-content name=''>...</text-content>
            <image-content name=''>...</image-content>
            ...
         </mms-contents>
      </mms>

   </delivery-channels>

</window-notification>
```

Notification windows use Simple Mail Transfer Protocol (SMTP) to send email, Short Message Service (SMS), and Multimedia Messaging Service (MMS) messages. To use these delivery channels, you must specify an **smtp** element to provide information about an SMTP server:

```
<smtp host='host' user='user' password='password' port='port' (opt, default='25') />
```

Only the **host** attribute of the element is required, because many SMTP servers run on the default port and do not require authentication:

```
<smtp host='mailhost.fyi.sas.com' />
```

However, it is a good practice to supply values for all the attributes of the **smtp** element:

```
<smtp host='smtp-server.ec.rr.com'
            user='esptest@ec.rr.com'
            password='esptest1' port='587' />
```

Notification Window Delivery Channels

Overview to Notification Window Delivery Channels

The notification window uses three types of delivery channel:

- **email** sends a multipart email message that contains text, HTML, and images to a specified email address
- **sms** sends an SMS message that contains text to an email address in the format **phoneNumber@gateway**
- **mms** sends a Multimedia Messaging Service (MMS) message that contains text and images to an email address in the format **phoneNumber@gateway**

Using the Email Delivery Channel

Here is XML code to use the **email** delivery channel:

```
<email throttle-interval='' test='true | false'>
   <deliver>[code]</deliver>
   <email-info>
      <sender>[code]</sender>
      <recipients>[code]</recipients>
      <subject>[code]</subject>
```

```
      <from>[code]</from>
      <to>[code]</to>
   </email-info>
   <email-contents>
      <text-content name=''>...</text-content>
      <html-content name=''>...</html-content>
      <image-content name=''>...</image-content>
      ...
   </email-contents>
</email>
```

The **email** element contains the following attributes:

- **throttle-interval** specifies a time period in which at most one notification is sent to a recipient
- **test** is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

The **deliver** element is optional. It contains a function to run in order to determine whether the notification should be sent.

The **email-info** element contains functions or hardcoded values that represent the data to be used to send an email notification. It contains the following elements:

- the **sender** email address
- the **recipients** to whom the email message is sent
- the **subject** of the email
- the **from** text of the email message
- the **to** text of the email
- The **email-contents** element, which contains the following elements:
  - the **text-content** element encloses the plain text content of the message
  - the **html-content** element encloses the HTML content of the message
  - the **image-content** element encloses a URL to image data

These elements can be interspersed in any way you want. The content of each element is included in the message in the order in which it appears. Any image data is retrieved and **base64** encoded before being inserted into the message.

Using the SMS Delivery Channel

Here is XML code to use the **sms** delivery channel:

```
<sms throttle-interval='' test='true | false'>
   <deliver>[code]</deliver>
   <sms-info>
      <sender>[code]</sender>
      <subject>[code]</subject>
      <from>[code]</from>
      <gateway>[code]</gateway>
      <phone>[code]</phone>
   </sms-info>
   <sms-contents>
      <text-content name=''>...</text-content>
```

```
   </sms-contents>
</sms>
```

The **sms** element contains the following attributes:

- **throttle-interval** specifies a time period in which at most one notification is sent a recipient.
- **test** is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

The **deliver** element is optional. It contains a function to run in order to determine whether the notification should be sent.

The **sms-info** element contains functions or hardcoded values that represent the data to be used to send an email notification. It contains the following elements:

- the **sender** email address.
- the **subject** of the email.
- the **from** text of the email message.
- the **gateway** element specifies the recipient's provider's SMS gateway. For example, AT&T is **txt.att.net**. Sprint is **messaging.sprintpcs.com**.
- the **sms-contents** element contains the body of the message to be sent. It contains the following element:
  - the **text-content** element encloses the plain text content of the message.

Using the MMS Delivery Channel

Here is XML code to use the MMS delivery channel:

```
<mms throttle-interval='' test='true | false'>
   <deliver>[code]</deliver>
   <mms-info>
      <sender>[code]</sender>
      <subject>[code]</subject>
      <gateway>[code]</gateway>
      <phone>[code]</phone>
   </mms-info>
   <mms-contents>
      <text-content name=''>...</text-content>
      <image-content name=''>...</image-content>
      ...
   </mms-contents>
</mms>
```

The **mms** element contains the following attributes:

- **throttle-interval** specifies a time period in which at most one notification is sent to a recipient.
- **test** is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

The **deliver** element is optional. It contains a function to run in order to determine whether the notification should be sent.

The **mms-info** element contains functions or hardcoded values that represent the data to be used to send an email notification. It contains the following elements:

- the **sender** email address.
- the **subject** of the email.
- the **gateway** element specifies the recipient's provider's SMS gateway. For example, AT&T is **txt.att.net**. Sprint is **messaging.sprintpcs.com**.
- the recipient **phone** number.
- the **mms-contents** element contains the body of the message to be sent. It contains the following elements:
  - the **text-content** element encloses the plain text content of the message.
  - the **image-content** element encloses a URL to image data.

These elements can be interspersed in any way you want. The content of each element is included in the message in the order it appears. Any image data is retrieved and **base64** encoded before being inserted into the message.

Using the Function-Context Element

The **function-context** element enables you to define functions to manipulate event data. You can use regular expressions, XML and XPath, or JSON to transform data from complex input information into usable entities.

Here is XML code that uses the function-context element:

```
<function-context>
   <expressions>
      <expression name=''>[Regular Expression]</expression>
      ...
   </expressions>
   <properties>
      <property-map name='' outer='' inner=''>[code]</property-map>
      <property-xml name=''>[code]</property-xml>
      <property-json name=''>[code]</property-json>
      <property-string name=''>[code]</property-string>
      <property-list name='' delimiter=''>[code]</property-list>
      <property-set name='' delimiter=''>[code]</property-set>
      ...
   </properties>
   <functions>
      <function name=''>[code]</function>
      ...
   </functions>
</function-context>
```

There are two types of functions that you can use:

- general functions (for example, **abs**, **ifNext**, and so on)
- functions specific to event stream processing (for example, **eventNumber**)

You can reference event fields in either the input event or the output event using the **$** notation, such as **$[*name_of_field*]**.

For example, suppose you have a name field in the input event and you want to generate an occupation field in the output event. In this case, you could use the following function:

```
<function name='occupation'>
ifNext
(
   equals($name,'larry'),'plumber',
   equals($name,'moe'),'electrician',
   equals($name,'curly'),'carpenter'
)
</function>
```

You could also reference fields in the output event. Continuing the above example, perhaps you want to add an **`hourlyWage`** to the output event that depends on occupation:

```
<function name='hourlyWage'>
ifNext
(
   equals($occupation,'plumber'),85.0,
   equals($occupation','electrician'),110.0,
   equals($occupation,'carpenter'),60.0
)
</function>
```

Note: Sequence is important when you define functions. When a function references an output event field, that field needs to be computed before the referring field

Expressions are POSIX regular expressions. Several functions are available to deal with regular expressions. Because regular expressions must be compiled before they can be used, the **`expressions`** element enables you to specify expressions to be compiled a single time when the function context is created. Then, the expression can be referenced from within functions using the following notation:

```
#[name_of_expression]
<function name='myData'>rgx(#myExpression,$inputField,1)
</function>
```

For example, suppose you receive a data field that contains a URI and you want to extract the protocol from it. If you use the following function:

```
<function name='protocol'>rgx('(.*):',$uri,1)
</function>
```

The regular expression is compiled each time that the function runs. However, if you use the following code:

```
<expressions>
   <expression name='getProtocol'>(.*):
   </expression>
</expressions>

<function name='protocol'>rgx(#getProtocol,$uri,1)
</function>
```

The expression is compiled a single time and used each time that the function runs.

Properties are similar to expressions in that they are referenced from within functions using the **`#`** notation: **`#[name_of_property]`**.

The **`properties`** element is a container for the following elements:

- **property-map** executes the function to generate a map of name-value pairs to be used for value lookups by name.
- **property-list** executes the function to generate a list of strings to be used for indexed access
- **property-set** executes the function to generate a set of strings to be used for value lookups
- **property-xml** executes the function to generate an XML object that can be used for XPath queries
- **property-json** executes the function to generate a JSON object that can be used for JSON lookups
- **property-string** executes the function to generate a string for general use in functions

Each property is generated using functions. These functions can reference properties defined before them in the XML.

For example, suppose you had employee information streaming into the model.

```
<event>
   <value name='map'>name:[employee name];position:[employee position]</value>
   <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
   <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>
```

You can use the **property-map** element to store employee data and examine the **position** field of the event in order to create a **property-xml** that contains the appropriate data. When the employee is a developer, the XML is created from **developerInfo**. Otherwise, it uses **managerInfo**.

The **function-context** element is specified as follows:

```
<function-context>
   <properties>
      <property-map name='myMap' outer=';' inner=':'>$map</property-map>
      <property-xml name='myXml'>
       if(equals(mapValue(#myMap,'position'),'developer'),
       $developerInfo,$managerInfo)</property-xml>
   </properties>
   <functions>
      <function name='employee'>mapValue(#myMap,'name')</function>
      <function name='info'>xpath(#myXml,'text()')</function>
   </functions>
</function-context>
```

When you stream the following events:

```
<event>
   <value name='map'>name:curly;position:developer</value>
   <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
   <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>

<event>
   <value name='map'>name:moe;position:manager</value>
   <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
   <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>
```

The **function-context** yields the following:

```
<event opcode='insert' window='project/query/transform'>
   <value name='employee'>curly</value>
   <value name='id'>fd26bf36-3d65-4d17-8dc6-317409bbf5b6</value>
   <value name='info'>this is developer info</value>
</event>

<event opcode='insert' window='project/query/transform'>
   <value name='employee'>moe</value>
   <value name='id'>84c56bb7-9f3c-4cb8-93a5-8dc2f75d353b</value>
   <value name='info'>this is manager info</value>
</event>
```

For more information about how to define each property, see "XML Language Elements Relevant to Notification Windows" on page 59.

Examples of Notification Windows

Building a Streaming Performance Monitor

Suppose that you want to stream a counter window into a filter window in order to check whether total throughput rate has dropped below 130,000 events per second. When that condition occurs, the event streams into a notification window that sends an SMS text message alerting someone of the slow streaming condition.

Here is the counter window:

```
<window-counter name='counter'
                       count-interval='2 seconds'
                       clear-interval='30 seconds'/>
```

It feeds the following filter window:

```
<window-filter name='poorPerformance'>
   <expression><![CDATA[totalSeconds > 10 and totalRate<130000]]></expression>
</window-filter>
```

Note that the **totalSeconds > 10**. You anticipate clocking slower rates as the data begins to stream.

Next, feed the event that indicates poor performance into a notification window:

```
<window-notification name='reportPerformance'>
   <smtp host='mailhost.fyi.sas.com' />
   <delivery-channels>
      <sms test='false' throttle-interval='2 hours'>
         <sms-info>
            <sender>brenda.doe@orion.com</sender>
            <from>ESP_Trade_Monitor</from>
            <subject>Slow Streaming</subject>
            <gateway>txt.att.net</gateway>
            <phone>5556466705</phone>
         </sms-info>
         <sms-contents>
            <text-content>
                The trade streaming has become very slow.
```

```
                    It is only processing $totalRate trades per second after running
                    for $totalSeconds seconds.
                </text-content>
            </sms-contents>
        </sms>
    </delivery-channels>
</window-notification>
```

You do not need extra schema or function context in this example. All the information you want to send is in the input event. The event generated by a counter window that looks like this:

```
<event opcode='upsert' window='project/query/counter'>
   <value name='input'>source</value>
   <value name='intervalCount'>283473</value>
   <value name='intervalRate'>141736</value>
   <value name='intervalSeconds'>2</value>
   <value name='totalCount'>782662</value>
   <value name='totalRate'>130444</value>
   <value name='totalSeconds'>6</value>
</event>
```

Grab the **`totalRate`** field and send that in an SMS along with the number of seconds that events have been streaming. The recipient gets an SMS message with this data.

Catching Front Running Traders

The following example catches stock traders when they attempt front running buys. A broker caught in the act is sent an email, an SMS message, and an MMS message. The message includes graphic details of the trades involved in the violation, and for the channels that permit graphics, contains an image of someone in a jail cell. All relevant message routing information is included in the broker dimension data:

```
i,n,1012112,Frodo,ESP,940 Orion Suite 201 Cary NC 27513,,frodo.doe@orion.com,5556466705,txt.att.net,mms.att.net
i,n,1012223,Sam,ESP,940 Orion Suite 201 Cary NC 27513,,sam.doe@orion.com,5556466706,txt.att.net,mms.att.net
i,n,1012445,Pippin,ESP,940 Orion Suite 201 Cary NC 27513,pippin.doe@orion.com,5556466707,txt.att.net,mms.att.net
i,n,1012334,Merry,ESP,940 Orion Suite 201 Cary NC 27513,merry.doe@orion.com,5556466708,txt.att.net,mms.att.net
i,n,101667,Gandalf,ESP,940 Orion Suite 201 Cary NC 27513,gandalf.doe@orion.com,5556466709,txt.att.net,mms.att.net
i,n,1012001,Aragorn,ESP,940 Orion Suite 201 Cary NC 27513,aragorn.doe@orion.com,5556466710,txt.att.net,mms.att.net
```

Note that the last four fields contain the email, phone number, and SMS and MMS gateways for each broker.

First, data streams into the model through a source window.

```
<window-source name='brokersSource' insert-only='true'>
   <schema-string>broker*:int32,brokerName:string,brokerage:string,
                  brokerAddress:string,brokerPhone:string,email:string,
                  smsGateway:string,mmsGateway:string</schema-string>
   <connectors>
      <connector class='fs'>
         <properties>
            <property name='type'>pub</property>
            <property name='fstype'>csv</property>
            <property name='fsname'>data/brokers.csv</property>
         </properties>
      </connector>
   </connectors>
</window-source>
```

A pattern window is constructed to detect front running violations. The pattern window needs to carry a lot of information because it deals with up to three trades. Each trade contains broker and customer information as well as the trade data. All of this data must be available to format a notification message.

The pattern window looks like this:

```
<window-pattern name='frontRunning'>
   <schema>
      <fields>
         <field name='id' type='int64' key='true' />
         <field name='broker' type='int32' />
         <field name='brokerName' type='string' />
         <field name='email' type='string' />
         <field name='phone' type='string' />
         <field name='sms' type='string' />
         <field name='mms' type='string' />
         <field name='customer' type='int32' />
         <field name='symbol' type='string' />
         <field name='tstamp1' type='string' />
         <field name='tstamp2' type='string' />
         <field name='tstamp3' type='string' />
         <field name='tradeId1' type='int32' />
         <field name='tradeId2' type='int32' />
         <field name='tradeId3' type='int32' />
         <field name='price1' type='double' />
         <field name='price2' type='double' />
         <field name='price3' type='double' />
         <field name='quant1' type='int32' />
         <field name='quant2' type='int32' />
         <field name='quant3' type='int32' />
         <field name='slot' type='int32' />
      </fields>
   </schema>
   <splitter-expr>
      <expression>slot</expression>
   </splitter-expr>
   <patterns>
      <pattern index='broker,symbol'>
         <events>
            <event name='e1'>((buysellflg == 1)
             and (broker == buyer)
             and (s == symbol)
             and (b == broker)
             and (p == price))</event>
            <event name='e2'>((buysellflg == 1)
             and (broker != buyer)
             and (s == symbol)
             and (b == broker))</event>
            <event name='e3'><![CDATA[((buysellflg == 0)
             and (broker == seller)
             and (s == symbol)
             and (b == broker)
             and (p < price))]]></event>
         </events>
         <logic>fby{1 hour}(fby{1 hour}(e1,e2),e3)</logic>
         <output>
```

```
            <field-selection name='broker' node='e1'/>
            <field-selection name='brokerName' node='e1'/>
            <field-selection name='brokerEmail' node='e1'/>
            <field-selection name='brokerPhone' node='e1'/>
            <field-selection name='brokerSms' node='e1'/>
            <field-selection name='brokerMms' node='e1'/>
            <field-selection name='buyer' node='e2'/>
            <field-selection name='symbol' node='e1'/>
            <field-selection name='date' node='e1'/>
            <field-selection name='date' node='e2'/>
            <field-selection name='date' node='e3'/>
            <field-selection name='id' node='e1'/>
            <field-selection name='id' node='e2'/>
            <field-selection name='id' node='e3'/>
            <field-selection name='price' node='e1'/>
            <field-selection name='price' node='e2'/>
            <field-selection name='price' node='e3'/>
            <field-selection name='quant' node='e1'/>
            <field-selection name='quant' node='e2'/>
            <field-selection name='quant' node='e3'/>
            <field-expr>1</field-expr>
         </output>
      </pattern>
      <pattern index='broker,symbol'>
         <events>
            <event name='e1'>((buysellflg == 0)
             and (broker == seller)
             and (s == symbol)
             and (b == broker))</event>
            <event name='e2'>((buysellflg == 0)
             and (broker != seller)
             and (s == symbol)
             and (b == broker))</event>
         </events>
         <logic>fby{10 minutes}(e1,e2)</logic>
         <output>
            <field-selection name='broker' node='e1'/>
            <field-selection name='brokerName' node='e1'/>
            <field-selection name='brokerEmail' node='e1'/>
            <field-selection name='brokerPhone' node='e1'/>
            <field-selection name='brokerSms' node='e1'/>
            <field-selection name='brokerMms' node='e1'/>
            <field-selection name='seller' node='e2'/>
            <field-selection name='symbol' node='e1'/>
            <field-selection name='date' node='e1'/>
            <field-selection name='date' node='e2'/>
            <field-expr> </field-expr>
            <field-selection name='id' node='e1'/>
            <field-selection name='id' node='e2'/>
            <field-expr>0</field-expr>
            <field-selection name='price' node='e1'/>
            <field-selection name='price' node='e2'/>
            <field-expr>0</field-expr>
            <field-selection name='quant' node='e1'/>
            <field-selection name='quant' node='e2'/>
            <field-expr>0</field-expr>
```

```
            <field-expr>2</field-expr>
         </output>
      </pattern>
   </patterns>
</window-pattern>
```

An event streams into the notification window.

```
<window-notification name='traderBusted'>
   <smtp host='smtp-server.ec.rr.com'
         user='esptest@ec.rr.com'
         password='esptest1' port='587' />
   <schema>
      <fields>
         <field name='id' type='int64' key='true' />
         <field name='broker' type='int32' />
         <field name='brokerName' type='string' />
         <field name='email' type='string' />
         <field name='phone' type='string' />
         <field name='sms' type='string' />
         <field name='mms' type='string' />
         <field name='customer' type='int32' />
         <field name='symbol' type='string' />
         <field name='tstamp1' type='string' />
         <field name='tstamp2' type='string' />
         <field name='tstamp3' type='string' />
         <field name='tradeId1' type='int32' />
         <field name='tradeId2' type='int32' />
         <field name='tradeId3' type='int32' />
         <field name='price1' type='double' />
         <field name='price2' type='double' />
         <field name='price3' type='double' />
         <field name='quant1' type='int32' />
         <field name='quant2' type='int32' />
         <field name='quant3' type='int32' />
         <field name='slot' type='int32' />
         <field name='day' type='string' />
         <field name='price1' type='double' />
         <field name='price2' type='double' />
         <field name='price3' type='double' />
         <field name='time1' type='string' />
         <field name='time2' type='string' />
         <field name='time3' type='string' />
         <field name='profit' type='double' />
      </fields>
   </schema>
   <function-context>
      <properties>
         <property-list name='time1' delimiter=' '>$tstamp1</property-list>
         <property-list name='time2' delimiter=' '>$tstamp2</property-list>
         <property-list name='time3' delimiter=' '>$tstamp3</property-list>
      </properties>
      <functions>
         <function name='profit'>product($quant3,diff($price3,$price1))</function>
         <function name='day'>listItem(#time1,0)</function>
         <function name='time1'>listItem(#time1,1)</function>
         <function name='time2'>listItem(#time2,1)</function>
```

```
      <function name='time3'>listItem(#time3,1)</function>
      <function name='price1'>precision($price1,2)</function>
      <function name='price2'>precision($price2,2)</function>
      <function name='price3'>precision($price3,2)</function>
   </functions>
</function-context>
<delivery-channels>
   <email test='true' throttle-interval='1 day'>
      <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
      <email-info>
         <sender>esptest@ec.rr.com</sender>
         <recipients>$email</recipients>
         <from>ESP Broker Surveillance</from>
         <to>$brokerName</to>
         <subject>You have been caught cheating, $brokerName</subject>
      </email-info>
      <email-contents>
         <html-content><![CDATA[
         <body>You bought <b>$quant1</b> shares of <b>$symbol</b>
               for $<b>$price1</b> on <b>$day</b> at <b>$time1</b>.
               You then bought <b>$symbol</b> for customer <b>$customer</b>
               at <b>$time2</b>, after which you sold <b>$quant3</b> shares of
               <b>$symbol</b> at <b>$time3</b> for $<b>$price3</b>,
               thus making you a profit of $<b>$profit</b>. <br/><br/></body>
         ]]></html-content>
         <image-content type='image'>
          http://esp-base:18080/esp/stuff/jail.jpg
         </image-content>
      </email-contents>
   </email>
   <mms test='true' throttle-interval='1 day'>
      <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
      <mms-info>
         <sender>esptest@ec.rr.com</sender>
         <subject>You have been caught cheating, $brokerName</subject>
         <gateway>$mms</gateway>
         <phone>$phone</phone>
      </mms-info>
      <mms-contents>
         <text-content>You bought $quant1 shares of $symbol
          for $$price1 on $day at $time1. You then bought $symbol for customer
          $customer at $time2, after which you sold $quant3 shares of $symbol
          at $time3 for $$price3, thus making you a profit of $$profit.
         </text-content>
         <image-content type='image'>
          http://esp-base:18080/esp/stuff/x.jpg
         </image-content>
      </mms-contents>
   </mms>
   <sms test='true' throttle-interval='1 day'>
      <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
      <sms-info>
         <sender>esptest@ec.rr.com</sender>
         <subject>You have been caught, $brokerName</subject>
         <from>ESP Broker Surveillance</from>
         <gateway>$sms</gateway>
```

```
            <phone>$phone</phone>
          </sms-info>
          <sms-contents>
            <text-content>You bought $quant1 shares of $symbol
            for $$price1 on $day at $time1. You then bought $symbol
            for customer $customer at $time2, after which you sold
            $quant3 shares of $symbol at $time3 for $$price3,
            thus making you a profit of $$profit.</text-content>
          </sms-contents>
        </sms>
      </delivery-channels>
    </window-notification>
```

Because this example uses MMS, you need to define a different SMTP server. Any email account referenced by that server must be specified in your SMTP configuration. The window calculates fields to use when formatting notification messages to the broker. A schema and a function context are defined.

When an event comes in, functions are run on the input event and schema data is created. You can use values from either the input event or the schema data in the message content. For example:

```
We noticed you bought <b>$quant1</b> shares of <b>$symbol</b> for $<b>$price1</b>
on <b>$day</b> at <b>$time1</b>. You then bought <b>$symbol</b> for
customer <b>$customer</b> at <b>$time2</b>, after which
you sold <b>$quant3</b> shares of <b>$symbol</b> at <b>$time3</b>
for $<b>$price3</b>, thus making you a profit of $<b>$profit</b>.
```

Note the number of variable references, some to the schema data (**`quant1`**, **`price1`**, **`price3`**, ...), and some to the input data (**`symbol`**). Variable references are also used to resolve the routing information for the notification:

```
<recipients>$email</recipients>
<gateway>$sms</gateway>
<phone>$phone</phone>
```

A function is used to determine when to send the notification. The same deliver function is used for all channels.

```
<deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
```

Whenever you see the notation **`@TOKEN@`** in an XML model, this means that the token is resolved when the project is loaded. These tokens can be resolved in one of three ways:

- on the command line, for example,**`dfesp_xml_server -BROKER pippin`**
- in your environment, for example, **`$ export BROKER=pippin`**
- in the properties for a project, for example, **`<property name='BROKER'>pippin</property>`**

In this case, you can specify which broker to use to send a notification.

Example Written in C++

```
dfESPwindow_notification  *notification =
       contquery->newWindow_notification("notify",NULL);
       notification->setSmtpConnection("mailhost.fyi.sas.com");

dfESPemail    *email = notification->addEmail();
email->setThrottleInterval("5 minutes");
```

```
email->setSender("brenda.doe@orion.com");
email->setRecipients("$email");
email->setSubject("Investment Opportunity");
email->setFrom("ESP");
email->setTo("Wealthy Trader");
email->addText("You traded $quant shares of $symbol at $$price.
               You must have a lot of money!");
email->setTestMode(true);

contquery->addEdge(joinBrokerData,0,notification);
```

Chapter 12

Creating Pattern Windows

Overview of Pattern Windows **175**
State Definitions for Operator Trees **178**
Restrictions on Patterns **180**
Using Stateless Pattern Windows **182**
Pattern Window Examples **182**
C++ Pattern Window Example 182
XML Pattern Window Examples 187

Overview of Pattern Windows

To create a pattern window, you do the following:

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

Specify EOIs by providing the following:

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

The valid logical operators for the pattern logic used by SAS Event Stream Processing are as follows:

Logical Operator	Function
`and`	All of its operands are true. Takes any number of operands.
`or`	Any of its operands are true. Takes any number of operands.
`fby`	Each operand is followed by the one after it. Takes any number of operands.

Logical Operator	Function
not	The operand is not true. Takes one operand.
notoccur	The operand never occurs. Takes one operand.
is	Ensure that the following event is as specified.

To apply a temporal condition to the **fby** function, append the condition to the function inside braces. For example, specify

```
fby{1 hour}(event1,event2)
```

when event2 happens within an hour of event1. Specify

```
fby{10 minutes}(event1,event2,event3)
```

when event3 happens within ten minutes of event2, and event2 happens within ten minutes of event1

Temporal conditions 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 this case, you must ensure that incoming events are in order with respect to the field-based date-time or timestamp.

Here is an XML example of a pattern from a broker surveillance model:

```
<pattern>
        <events>
            <event name='e1'>((buysellflg==1) and (broker == buyer)
            and (s == symbol) and (b == broker) and (p == price))</event>
            <event name='e2'>((buysellflg==1) and (broker != buyer)
            and (s == symbol) and (b == broker))</event>
            <event name='e3'><![CDATA[((buysellflg==0) and (broker == seller)
            and (s == symbol) and (b == broker) and (p < price))]]></event>
        </events>
        <logic>fby{1 hour}(fby{1 hour}(e1,e2),e3)</logic>
        ...
        </output>
    </pattern>
    <pattern>
        <events>
            <event name='e1'>((buysellflg==0) and (broker == seller)
            and (s == symbol) and (b == broker))</event>
            <event name='e2'>((buysellflg==0) and (broker != seller)
            and (s == symbol) and (b == broker))</event>
        </events>
        <logic>fby{10 minutes}(e1,e2)</logic>
        ...
    </pattern>
</patterns>
</window-pattern>
```

Here is an XML example of a pattern from an e-commerce model:

```
<pattern>
        <events>
            <event name='e1'>eventname=='ProductView'
            and c==customer and p==product</event>
```

```
            <event name='e2'>eventname=='AddToCart'
            and c==customer and p==product</event>
            <event name='e3'>eventname=='CompletePurchase'
            and c==customer</event>
            <event name='e4'>eventname=='Sessions'
            and c==customer</event>
            <event name='e5'>eventname=='ProductView'
            and c==customer and p!=product</event>
            <event name='e6'>eventname=='EndSession'
            and c==customer</event>
        </events>
        <logic>fby(e1,fby(e2,not(e3)),e4,e5,e6)</logic>
        ...
    </pattern>
```

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

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

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

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

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

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

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

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

When you want to apply a temporal condition to the FBY operator, append the condition to the function inside braces. For example:

```
    fby{1 hour}(event1,event2)
fby{10 minutes}(event1,event2,event3)
```

In the first line of code, event2 happens within an hour of event1. In the second line, event3 happens within ten minutes of event2, which happens within ten minutes of event1.

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

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

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

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

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

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

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

State Definitions for Operator Trees

Operator trees can have one of the following states:

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

The state value of an operator sub-tree can be FIXED or not-FIXED. When the state value is FIXED, no further events should be applied to it. When the state value is not-

FIXED, the state value could change based on application of an event. New events should be applied to the sub-tree.

When a pattern instance fails to emit a match and destroys itself, it folds. The instance is freed and removed from the active pattern instance list. When the top-level tree in a pattern instance (the root node) becomes FALSE, the pattern folds. When it becomes TRUE, the pattern emits a match and destroys itself.

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

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

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

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

notoccur *OPT*
: this is not allowed

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

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

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

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

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

is *EOI*
: becomes TRUE on the application of an event that satisfies the EOI and FALSE conditions. Becomes FIXED on the first application of an event.

is *OPT*
this is not allowed

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

Restrictions on Patterns

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

- The data type of the key field must be **`int64`**.
- 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, e2,2880000000l);
  ```

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

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

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

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

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

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

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

Using Stateless Pattern Windows

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

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

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

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

Pattern Window Examples

C++ Pattern Window Example

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

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

// Include class definitions for modeling objects.
//
#include "dfESPwindow_source.h"
#include "dfESPwindow_pattern.h"
#include "dfESPevent.h"
#include "dfESPcontquery.h"
#include "dfESPengine.h"
#include "dfESPproject.h"

using namespace std;

// Declare a context data structures to ensure the
//    callback function thread safe and set window name
```

```
//
struct callback_ctx {
    dfESPthreadUtils::mutex *lock;
    dfESPstring windowName;
    };

//    This is a simple callback function that can be registered
//    for a window's 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.  In this case
//    the context structure is used to ensure that the function is
//    thread safe.
//
void winSubscribeFunction(dfESPschema *os, dfESPeventblockPtr ob, void *cntx) {
    callback_ctx *ctx = (callback_ctx *)cntx;

    ctx->lock->lock();
    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 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
    ctx->lock->unlock();
    }

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 output,
    //          -b <badevent.name> to write any bad events (events that failed
    //             to be applied to a window index).
    //          -r <restore.path> path used to restore a previously persisted
    //             engine state.
    // @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: dfESPLLInfo,
```

```
//        @see dfESPLoggingLevel
// @param logConfigFile a log4SAS configuration file
//        - default: configure logging to go to standard out.
// @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 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");

// 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, 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",  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",  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 temporal check
}

// Add the subscriber callback to the pattern window.
// Callback context structure is used to ensure the function
// thread safe.
callback_ctx pattern_ctx;
pattern_ctx.lock = dfESPthreadUtils::mutex::mutex_create();
// create the lock
pw_01->addSubscriberCallback(winSubscribeFunction, (void *)&pattern_ctx);

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

// Define the project's thread pool size and start it.
//
// **Note** after we start the project here, we do not see
//    anything happen, as no data has yet been put into the
//    continuous query.
//

project_01->setNumThreads(2);
myEngine->startProjects();

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

//
// At this point the project is running in the background using
// the defined thread pool. We'll use the main thread that
// we are in to inject some data.


// Generate some test event data and inject it into the source window.
bool eventFailure;
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",
                   eventFailure);
if (eventFailure) {
    cerr << "Creating event failed. Aborting..." << endl;
    abort();
    }
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",
                    eventFailure);
        if (eventFailure) {
            cerr << "Creating event failed. Aborting..." << endl;
            abort();
            }
        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",
                    eventFailure);
        if (eventFailure) {
            cerr << "Creating event failed. Aborting..." << endl;
            abort();
            }
        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",
                    eventFailure);
        if (eventFailure) {
            cerr << "Creating event failed. Aborting..." << endl;
            abort();
            }
        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",
                    eventFailure);
        if (eventFailure) {
            cerr << "Creating event failed. Aborting..." << endl;
            abort();
            }
        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 system stops processing before shutting down

        // Now shutdown.
        myEngine->shutdown();
        return 0;
    }
```

After you execute this code, you obtain these results:

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

XML Pattern Window Examples

```
<window-pattern name='front_running'>
  <schema-string>
   id*:int64,broker:int32,brokerName:string,typeFlg:int32,symbol:
       string,tstamp1:date,tstamp2:date,tstamp3:date,tradeId1:
       int32,tradeId2:int32,tradeId3:int32
  </schema-string>
  <patterns>
    <pattern>
      <events>
        <event name='e1'>((buysellflg==1) and (broker == buyer)
                          and (s == symbol) and (b == broker)
                          and (p == price))
        </event>
        <event name='e2'>((buysellflg==1) and (broker != buyer)
                          and (s == symbol) and (b == broker))
        </event>
        <event name='e3'><![CDATA[
          buysellflg==0) and (broker == seller)
          and (s == symbol) and (b == broker)
          and (p < price))]]></event>
      </events>
      <logic>fby(e1,e2,e3)</logic>
      <output>
       <field-selection name='broker' node='e1'/>
       <field-selection name='brokerName' node='e1'/>
       <field-expr>1</field-expr>
       <field-selection name='symbol' node='e1'/>
       <field-selection name='date' node='e1'/>
       <field-selection name='date' node='e2'/>
       <field-selection name='date' node='e3'/>
       <field-selection name='id' node='e1'/>
       <field-selection name='id' node='e2'/>
       <field-selection name='id' node='e3'/>
      </output>
     </pattern>
    <pattern>
      <events>
        <event name='e1'>((buysellflg==0) and (broker == seller)
                          and (s == symbol) and (b == broker))
        </event>
        <event name='e2'>((buysellflg==0) and (broker != seller)
                          and (s == symbol) and (b == broker))
        </event>
      </events>
      <logic>fby(e1,e2)</logic>
      <output>
        <field-selection name='broker' node='e1'/>
        <field-selection name='brokerName' node='e1'/>
        <field-expr>2</field-expr>
```

```
                              <field-selection name='symbol' node='e1'/>
                              <field-selection name='date' node='e1'/>
                              <field-selection name='date' node='e2'/>
                              <field-expr> </field-expr>
                              <field-selection name='id' node='e1'/>
                              <field-selection name='id' node='e2'/>
                              <field-expr>0</field-expr>
                            </output>
                           </pattern>
                          </patterns>
                        </window-pattern>

<window-pattern name='sigma2Pattern_calc' index='pi_EMPTY' >
          <schema>
             <fields>
                <field name='alertID' type='int64' key='true'/>
                <field name='ID1' type='int64'/>
                <field name='element1' type='string'/>
                <field name='attribute1' type='string'/>
                <field name='timestamp1' type='stamp'/>
                <field name='value1' type='double'/>
                <field name='valueAve1' type='double'/>
                <field name='valueMin1' type='double'/>
                <field name='valueMax1' type='double'/>
                <field name='valueStd1' type='double'/>
                <field name='ID2' type='int64'/>
                <field name='element2' type='string'/>
                <field name='attribute2' type='string'/>
                <field name='timestamp2' type='stamp'/>
                <field name='value2' type='double'/>
                <field name='valueAve2' type='double'/>
                <field name='valueMin2' type='double'/>
                <field name='valueMax2' type='double'/>
                <field name='valueStd2' type='double'/>
             </fields>
          </schema>
          <patterns>
             <pattern index='element,attribute'>
                <events>
                   <event source='original2stdDevCheck' name='e1'>
                     (value&lt;(valueAve-2*valueStd))
                                and (r_cnt==elementReadingCount)
                                and (pid==element and aid==attribute)
                   </event>
                   <event source='original2stdDevCheck' name='e2'>
                     (pid==element and aid==attribute)
                      and (r_cnt&gt;=elementReadingCount-2)
                      and (value&lt;(valueAve-2*valueStd))
                   </event>
                </events>
                <logic>fby{18600000001 seconds}(e1, e2)</logic>
                <output>
                   <field-selection name='sequence' node='e1'/>
                   <field-selection name='element' node='e1'/>
                   <field-selection name='attribute' node='e1'/>
                   <field-selection name='timestamp' node='e1'/>
```

```
            <field-selection name='value' node='e1'/>
            <field-selection name='valueAve' node='e1'/>
            <field-selection name='valueMin' node='e1'/>
            <field-selection name='valueMax' node='e1'/>
            <field-selection name='valueStd' node='e1'/>
            <field-selection name='sequence' node='e2'/>
            <field-selection name='element' node='e2'/>
            <field-selection name='attribute' node='e2'/>
            <field-selection name='timestamp' node='e2'/>
            <field-selection name='value' node='e2'/>
            <field-selection name='valueAve' node='e2'/>
            <field-selection name='valueMin' node='e2'/>
            <field-selection name='valueMax' node='e2'/>
            <field-selection name='valueStd' node='e2'/>
         </output>
      </pattern>
   </patterns>
</window-pattern>
```

```
              <pattern>
                      <events>
                          <event name='e1'>eventname=='ProductView'
                                    and c==customer and p==product</event>
                          <event name='e2'>eventname=='AddToCart'
                                    and c==customer and p==product</event>
                          <event name='e3'>eventname=='CompletePurchase'
                                    and c==customer</event>
                          <event name='e4'>eventname=='Sessions'
                                    and c==customer</event>
                          <event name='e5'>eventname=='ProductView'
                                    and c==customer and p!=product</event>
                          <event name='e6'>eventname=='EndSession'
                                    and c==customer</event>
                      </events>
                      <logic>fby(e1,fby(e2,not(e3)),e4,e5,e6)</logic>
                      ...
```

Chapter 13

Creating Procedural Windows

Overview to Procedural Windows **191**

Using C++ Window Handlers **192**

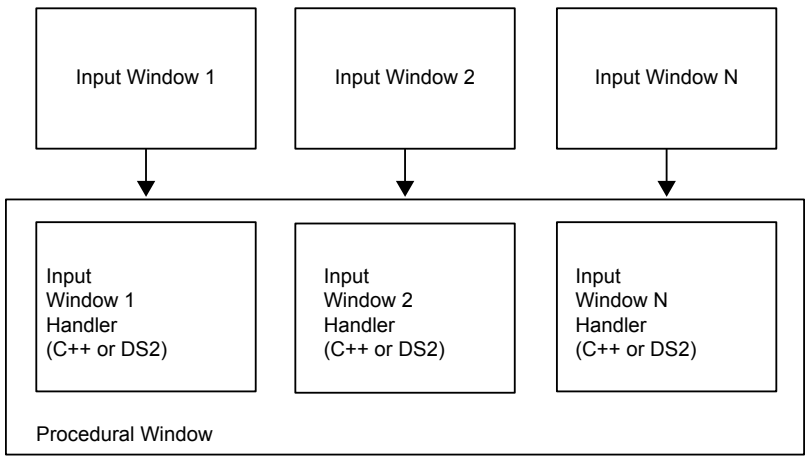
Using DS2 Window Handlers **194**
Overview of DS2 Window Handlers 194
General Structure of a DS2 Input Handler 195
Examples 195
Event Stream Processor to DS2 Data Type Mappings and Conversions 197

XML Examples of Procedural Windows **198**

Overview to Procedural Windows

A procedural window enables the specification of an arbitrary number of input windows and input handler functions for each input window. You can write procedural window input handlers in C++, DS2, or XML. When an input event arrives, the input handler registered for the matching window is called. The events produced by this input handler function are then output.

Figure 13.1 *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.

Using 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");

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",
                             dfESPindextypes::pi_HASH,
                             source_sch);

dfESPwindow_procedural *pw;
pw = cq_01->newWindow_procedural("procedural window",
                                 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.

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 permit models to perform data quality checks and log events that do not pass. There are two common reasons to reject an event:

- 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).

Using DS2 Window Handlers

Overview of DS2 Window Handlers

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

Here is a simple example:

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

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

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

All fields of the input window are seen as variables in DS2 programs, so can be used in calculations. The variable _opcode is available and takes the integer values **1** (Insert), **2** (Update), **3**(Delete), **4** (Upsert), or **5** (Safe Delete). The variable _flag is available and takes the integer values **1** (Normal) or **3**(Retention). The variables exported from the DS2 program are all the input variables plus any global variables declared. This set of variables is then filtered by the schema field names of the procedural window to form the output event.

You can retain the state of a variable in a DS2 program. The state remains valid during the life of the project. For example, the **retain** statement in the following DS2 data block makes the **sequence** variable static, maintaining state from call to call. The variable is tracked between rows within an event block and across event blocks.

```
<ds2-code source='MMD1'>
   <![CDATA[
       ds2_options cdump;
       data esp.out;
          dcl integer sequence;
          retain sequence 0;
           method run();
              set esp.in;
              sequence = sequence + 1;
           end;
       enddata;
```

```
  ]]>
 </ds2-code>
```

General Structure of a DS2 Input Handler

DS2 input handlers use the following boilerplate definition:

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

Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            ID = ID + 1;
            output;
        end;
    enddata;
```

Given this input:

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

The following output is produced:

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

Event Stream Processor to DS2 Data Type Mappings and Conversions

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

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

From/To	Integer	BigInt	Double	Date	Time	Timestamp	Char	Varchar
timestamp				x	x	x		
utf8str							x	x

XML Examples of Procedural Windows

You can write procedural windows in XML using the **window-procedural** element. For more information about this element, see Chapter 3, "Using the XML Layer,".

```
<window-procedural name='pw_01'>
  <context-plugin name='libmethod' function='get_derived_context'/>
  <schema>
    <fields>
      <field name='ID' type='int32' key='true' />
      <field name='insertCount' type='int32' />
      <field name='updateCount' type='int32'/>
      <field name='deleteCount' type='int32'/>
    </fields>
  </schema>
  <plugin source='sourceWindow_01' name='libmethod' function='countOpcodes'/>
    <connectors>
      <connector class='fs'>
        <properties>
          <property name='type'>sub</property>
          <property name='fstype'>csv</property>
          <property name='fsname'>procedural1.csv</property>
          <property name='snapshot'>true</property>
        </properties>
     </connector>
    </connectors>
</window-procedural>

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

```
                    _opcode = 4;
                end;
            enddata;
    ]]>
  </ds2-code>
</window-procedural>
```

Chapter 14

Advanced Window Operations

Implementing Periodic (or Pulsed) Window Output **201**

Splitting Generated Events across Output Slots **202**
Overview 202
Splitter Functions 202
Splitter Expressions 203

Marking Events as Partial-Update on Publish **204**
Overview 204
Publishing Partial Events into a Source Window 204
Examples 205

Understanding Retention **206**

Understanding Primary and Specialized Indexes **208**
Overview 208
Fully Stateful Indexes 209
Using the pi_HLEVELDB Primary Index with Big Dimension Tables 210
Non-Stateful Index 214

Persist and Restore Operations **215**

Gathering and Saving Latency Measurements **217**

Implementing Periodic (or Pulsed) Window Output

In most cases, the SAS Event Stream ProcessingAPI is fully event driven. That is, windows continuously produce output as soon as they transform input. But there might be times when you 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 a single operation.

Here are two cases where batched output might be useful:

- Visualization clients might want to get updates once a second because 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 that follows the pulsed window is interested in comparing the deltas between periodic snapshots from that window.

Use the following call to add output pulsing to a window:

```
dfESPwindow::setPulseInterval(size_t us);
```

Note: Periodicity is specified in microseconds. However, given the clock resolution of most non-real-time operating systems, the minimum value that you should specify for a pulse period is 100 milliseconds.

Splitting Generated Events across Output Slots

Overview

All window types can register a splitter function or expression to determine what output slot or slots should be used for a newly generated event. This enables you to send generated events across a set of output slots.

Most windows send all generated events out of output slot 0 to zero of more downstream windows. For this reason, it is not standard for most models to use splitters. Using window splitters can be more efficient than using filter windows off a single output slot. This is especially true, for example, when you are performing an alpha-split across a set of trades or a similar task.

Using window splitters is more efficient than using two or more subsequent filter windows. This is because the filtering is performed a single time at the window splitter rather than multiple times for each filter. This results in less data movement and processing.

Splitter Functions

Here is a prototype for a splitter function.

```
size_t splitterFunction(dfESPschema *outputSchema, dfESPeventPtr nev,
     dfESPeventPtr oev);
```

This splitter function receives the schema of the events supplied, the new and old event (only non-null for update block), and it returns a slot number.

Here is how you use the splitter for the source window (**`sw_01`**) to split events across three copy windows: **`cw_01, cw_02, cw_03`**.

```
sw_01->setSplitter(splitterFunction);
cq_01->addEdge(sw_01, 0,  cw_01);
cq_01->addEdge(sw_01, 1,  cw_02);
cq_01->addEdge(sw_01, -1, cw_03);
```

The **`dfESPwindow::setSplitter()`** member function is used to set the user-defined splitter function for the source window. The **`dfESPcontquery::addEdge()`** member function is used to connect the copy windows to different output slots of the source window.

When adding an edge between windows in a continuous query, specify the slot number of the parent window where the receiving window receives its input events. If the slot number is -1, it receives all the data produced by the parent window regardless of the splitter function.

If no splitter function is registered with the parent window, the slots specified are ignored, and each child window receives all events produced by the parent window.

Note: Do not write a splitter function that randomly distributes incoming records. Also, do not write a splitter function that relies on a field in the event that might change. The change might cause the updated event to generate a different slot value than what was produced prior to the update. This can cause an Insert to follow one path and a subsequent Update to follow a different path. This generates inconsistent results, and creates indices in the window that are not valid.

Splitter Expressions

When you define splitter expressions, you do not need to write the function to determine and return the desired slot number. Instead, the registered expression does this using the splitter expression engine. Applying expressions to the previous example would look as follows, assuming that you split on the field name "splitField", which is an integer:

```
sw_01->setSplitter("splitField%2");
cq_01->addEdge(sw_01, 0,  cw_01);
cq_01->addEdge(sw_01, 1,  cw_02);
cq_01->addEdge(sw_01, -1, cw_03);
```

Here, the **`dfESPwindow::setSplitter()`** member function is used to set the splitter expression for the source window. Using splitter expressions rather than functions can lead to slower performance because of the overhead of expression parsing and handling. Most of the time you should not notice differences in performance.

`dfESPwindow::setSplitter()` has two additional optional parameters with defaults set to NULL.

- **`initExp`** enables you to specify an initialization expression for the expression engine used for this window's splitter.
- **`initRetType`** enables you to specify a return **`datavar`** value in those cases when you want to pass state from the initialization expression to the C++ application thread that makes the call. Most initialization expressions do not use return values from the initialization.

This initialization message enables you to specify some setup state, perhaps variable declarations and initialization, that you can use later in the splitter expression processing.

The full syntax for this call is as follows:

```
dfESPdatavarPtr setSplitter(const char* splitterExp, const char*
                            initExp=NULL, dfESPdatavar::dfESPdatatype
                            initRetType=dfESPdatavar::ESP_NULL);
```

You can find an example of window output splitter initialization in **`splitter_with_initexp`** in **`$DFESP_HOME/examples/cxx`**. The example uses the following **`setSplitter`** call where the initialize declares and sets an expression engine variable to 1:

```
(void)sw_01->setSplitter("counter=counter+1; return counter%2",
                                        "integer counter\r\ncounter=1");
```

For each new event the initialize increments and mods the counter so that events rotate between slots 0 and 1.

Marking Events as Partial-Update on Publish

Overview

In most cases, events are published into an 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 SAS Event Stream Processing to match key fields of an event retained in the system with the current event and not to update the current event's fields.

In order for a published event to be tagged as a partial-update, the event must contain all non-null key fields that match an existing event in the source window. Partial updates are applied to source windows only.

When using transactional event blocks that include partial events, be careful that all partial updates are for key fields that are already in the source window. You cannot include the insert of the key values with an update to the key values in a single event block with transactional properties. This attempt fails and is logged because transactional event blocks are treated atomically. All operations in that block are checked against an existing window state before the transactional block is applied as a whole.

Publishing Partial Events into a Source Window

Consider these three points when you publish partial events into a source window.

- In order to construct the partial event, you must represent all the fields in the event. Specify either the field type and value or a placeholder field that indicates that the field value and type are missing. In this way, the existing field value for this key field combination remains for the updated event. These field values and types can be provided as **datavars** to build the event. Alternatively, they can be provided as a comma-separated value (CSV) string.

  If you use CSV strings, then use '^U' (such as, control-U, decimal value 21) to specify that the field is a placeholder field and should not be updated. On the other hand, if you use **datavars** to represent individual fields, then those fully specified fields should be valid. Enter them as **datavars** with values (non-null or null). Specify the placeholder fields as empty **datavars** of type **dfESPdatavar::ESP_LOOKUP**.

- No matter what form you use to represent the field values and types, the representation should be included in a call for the partial update to be published. In addition to the fields, use a flag to indicate whether the record is a normal or partial update. If you specify partial update, then the event must be an Update or an Upsert that is resolved to an Update. Using partial-update fields makes sense only in the context of updating an existing or retained source window event. This is why the

opcode for the event must resolve to Update. If it does not resolve to Update, an event merge error is generated.

If you use an event constructor to generate this binary event from a CSV string, then the beginning of that CSV string contains "**u,p**" to show that this is a partial-update. If instead, you use **event->buildEvent()** to create this partial update event, then you need to specify the event flag parameter as **dfESPeventcodes::ef_PARTIALUPDATE** and the event opcode parameter as **dfESPeventcodes::eo_UPDATE**.

- One or more events are pushed onto a vector and then that vector is used to create the event block. The event block is then published into a source window. For performance reasons, each event block usually contains more than a single event. When you create the event block, you must specify the type of event block as transactional or atomic using **dfESPeventblock::ebt_TRANS** or as normal using **dfESPeventblock::ebt_NORMAL**.

  Do not use transactional blocks with partial updates. Such usage treats all events in the event block as atomic. If the original Insert for the event is in the same event block as a partial Update, then it fails. The events in the event block are resolved against the window index before the event block is applied atomically. Use normal event blocks when you perform partial Updates.

Examples

Here are some sample code fragments for the variations on the three points described in the previous section.

Create a partial Update **datavar** and push it onto the **datavar** vector.

```
// Create an empty partial-update datavar.
dfESPdatavar* dvp  = new dfESPdatavar(dfESPdatavar::ESP_LOOKUP);
// Push partial-update datavar onto the vector in the appropriate
// location.
// Other partial-update datavars might also be allocated and pushed to the
// vector of datavars as required.
dvVECT.push_back(dvp); // this would be done for each field in the update
     event
```

Create a partial Update using partial-update and normal **datavars** pushed onto that vector.

```
// Using the datavar vector partially defined above and schema,
// create event.
dfESPeventPtr eventPtr = new dfESPevent();
eventPtr->buildEvent(schemaPtr, dvVECT, dfESPeventcodes::eo_UPDATE,
dfESPeventcodes::ef_PARTIALUPDATE);
```

Define a partial update event using CSV fields where '**^U**' values represent partial-update fields. Here you are explicitly showing '**^U**'. However, in actual text, you might see the character representation of **Ctrl-U** because individual editors show control characters in different ways.

Here, the event is an Update (due to '**u**'), which is partial-update (due to '**p**'), key value is 44001, "ibm" is the instrument that did not change. The instrument is included in the field. The price is 100.23, which might have changed, and 3000 is the quantity, which might have changed, so the last three of the fields are not updated.

```
p = new dfESPevent(schema_01,
(char *)"u,p,44001,ibm,100.23,3000,^U,^U,^U");
```

Understanding Retention

Any source or copy window can set a retention policy. A window's retention policy governs how it introduces Deletes into the event stream. These Deletes work their way along the data flow, recomputing the model along the way. Internally generated Deletes are flagged with a retention flag, and all further window operations that are based on this Delete are flagged.

For example, consider a source window with a sliding volume-based retention policy of two. That source window always contains at most two events. When an Insert arrives causing the source window to grow to three events, the event with the oldest modification time is removed. A Delete for that event is executed.

Retention Type	Description
time-based	Retention is performed as a function of the age of events. The age of an event is calculated as current time minus the last modification time of the event. Time can be driven by the system time or by a time field that is embedded in the event. A window with time-based retention uses current time set by the arrival of an event.
volume-based	Retention is based on a specified number of records. When the volume increases beyond that specification, the oldest events are removed.

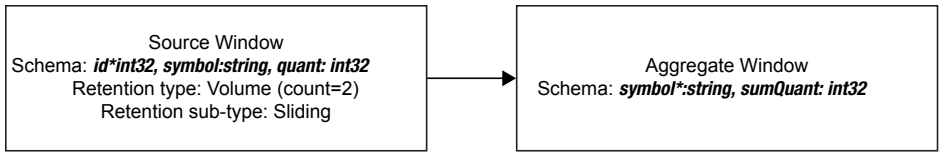
Both time and volume-based retention can occur in one of two variants:

Retention Variant	Description
sliding	Specifies a continuous process of deleting events. Think of the retention window sliding continuously. For a volume-based sliding window, when the specified threshold is hit, one delete is executed for each insert that comes in.
jumping	Specifies a window that completely clears its contents when a specified threshold value is hit. Think of a ten-minute jumping window as one that deletes its entire contents every 10 minutes.

A canonical set of events is a collapsed minimal set of events such that there is at most one event per key. Multiple updates for the same key and insert + multiple updates for the same key are collapsed. A window with retention generates a canonical set of changes (events). Then it appends retention-generated Deletes to the end of the canonical event set. At the end of the process, it forms the final output block.

Windows with retention produce output event blocks of the following form: **`{<canonical output events>, <canonical retention deletes>}`**. All other windows produce output blocks of the following form: **`{<canonical output events>}`**.

Consider the following model:



The following notation is used to denote events **[<opcode>/<flags>: f1, ... ,fn]**

- Opcode
  - **i** — insert
  - **d** — delete
  - **ub** — update block — any event marked as **ub** is always followed by an event marked as **d**
- Flags
  - **n** — normal
  - **r** — retention generated

Suppose that the following events are streamed into the model:

Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 1,ibm,10]	**[i/n: 1,ibm,10]**	**[i/n: ibm,10]**
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 2,ibm,11]	**[i/n: 2,ibm,11]**	**[ub/n: ibm,21]** **[d/n: ibm,10]**
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 3,sas,100]	**[i/n: 3,sas,100]**	**[i/n: sas,100]**
	[d/r: 1,ibm,10]	**[ub/r: ibm,11]** **[d/r: ibm,21]**
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 4,ibm,12]	**[i/n: 4,ibm,12]**	**[ub/r: ibm,12]**
	[d/r: 2,ibm,11]	**[d/r: ibm,11]**

When you run in retention-tracking mode, retention and non-retention changes are pulled through the system jointly. When the system processes a user event, the system generates a retention Delete. Both the result of the user event and the result of the retention Delete are pushed through the system. You can decide how to interpret the

result. In normal retention mode, these two events can be combined to a single event by rendering its output event set canonical.

Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 1,ibm,10]	**[i/n: 1,ibm,10]**	**[i/n: ibm,10]**
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 2,ibm,11]	**[i/n: 2,ibm,11]**	**[ub/n: ibm,21]** **[d/n: ibm,10]**
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 3,sas,100]	**[i/n: 3,sas,100]**	**[i/n: sas,100]**
	[d/r: 1,ibm,10]	**[ub/r: ibm,11]** **[d/r: ibm,21]**
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 4,ibm,12]	**[i/n: 4,ibm,12]**	**[ub: ibm,23]** **[d/n: ibm,11]**
	[d/r: 2,ibm,11]	**[ub/r: ibm,12]** **[d/r: ibm,23]**

Here, the output of the aggregate window, because of the last input event, is non-canonical. In retention tracking mode, you can have two operations per key when the input events contain a user input for the key and a retention event for the same key.

Note: A window with pulsed mode set always generates a canonical block of output events. For the pulse to function as designed, the window buffers output events until a certain threshold time. The output block is rendered canonical before it is sent.

Understanding Primary and Specialized Indexes

Overview

In order to process events with opcodes, each window must have a primary index. That index enables the rapid retrieval, modification, or deletion of events in the window.

Windows can have other indexes that serve specialized purposes.

- source and copy windows have an additional index to aid in retention
- aggregate windows have an aggregation index to maintain the group structure
- Join windows have left and right local indexes along with optional secondary indexes. These help avoid locking and maintain data consistency.

Table 14.1 *Index Types Associated with Each Window Type*

Window Type	Primary Index	Retention Index	Aggregation Index	Left Local Index	Right Local Index
Filter Window Compute Window Pattern Window Procedural Window Textcontext Window	Yes				
Source Window Copy Window	Yes	Yes			
Aggregate Window	Yes		Yes		
Join Window	Yes			Yes Optional secondary	Yes Optional secondary

The **`dfESPeventdepot`** object that is used to store windows supports six types of primary indices: five are stateful, and one is not.

Fully Stateful Indexes

The following index types are fully stateful:

Index Type	Description
`pi_RBTREE`	Specifies red-black tree, logarithmic Insert, Deletes are O(log(n)) — provides smooth latencies. Stores events in memory.
`pi_HASH`	Specifies a typical open hash algorithm. Stores events in memory. In general this index provides faster results than **`pi_RBTREE`**. Unless properly sized, using this index might lead to latency spikes.
`pi_CL_HASH`	Specifies a closed hash. This index provides faster results than **`pi_HASH`**.
`pi_FW_HASH`	Specifies a forward hash. This index creates a smaller memory footprint than other hash indexes, but might yield poorer delete performance.
`pi_LN_HASH`	Specifies a linked hash. This index performs slightly more slowly than other hash index and uses more memory than **`pi_CL_HASH`**.

Index Type	Description
`pi_HLEVELDB`	On disk stateful index for large source or copy windows and for the left or right local dimension index in a join. Can be used when there is no retention, aggregation, or a need for a secondary index

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.

Using the pi_HLEVELDB Primary Index with Big Dimension Tables

Overview

A common use case is to stream fact data into an engine and join it with dimension data for further processing. This works well under the following two conditions:

- The model is stateless or is controlled by retention.
- The entire dimension table fits into memory.

However, when the dimension table consists of tens or hundreds of million rows, this common use case becomes problematic. You can increase system memory to an extent, after which price and hardware limitations affect the size of the data that can be effectively processed.

With massive dimension tables, you can use the `pi_HLEVELDB` index to store events in an on-disk store. This produces large on-disk event indexes with a correspondingly small RAM footprint. Using this index type is helpful in the following circumstances:

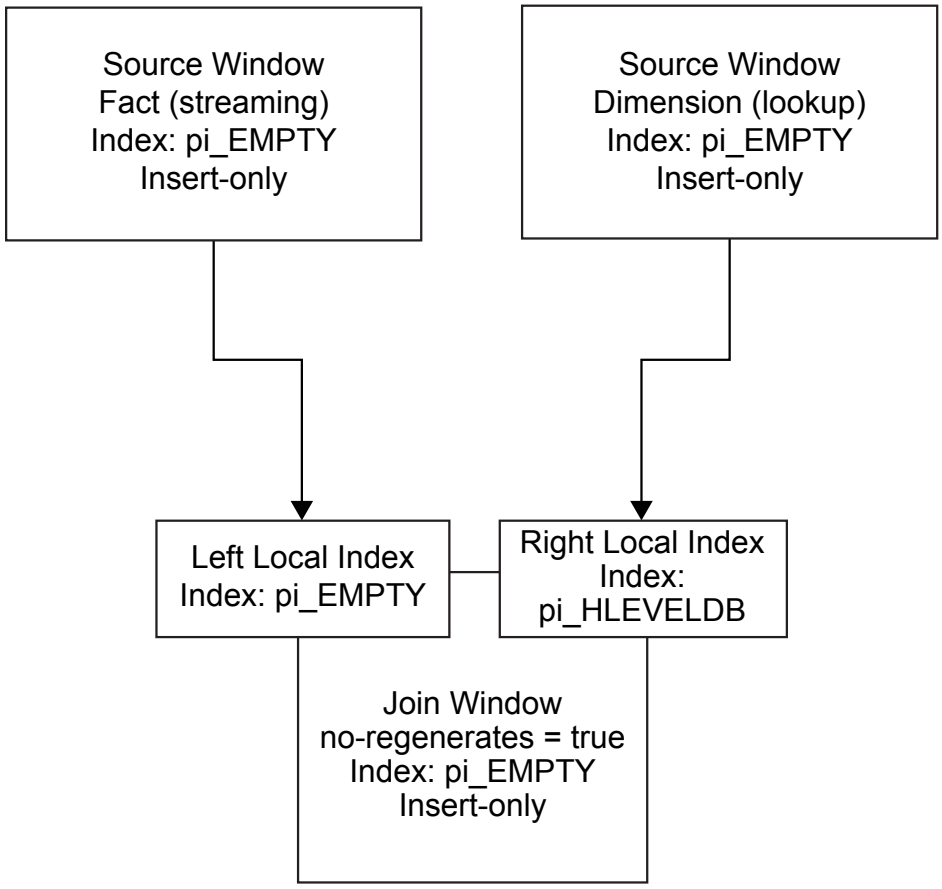
- Throughput is measured in tens of thousands of events per second or more.
- The window is not implementing retention or aggregation.
- No secondary index is used.

Stateless Left Outer Join: One-Time Bulk Load with No-Regeneration

Consider the following case. A stateless left outer join streams Insert-only data through the left window (fact) and matches it against dimensional data on the right. It passes Inserts out of the join. It uses the `no-regenerates` option of the join window, so future inserts to the dimension table affect only future streaming fact events.

In this model, you first prime the dimension table with a large volume of Inserts, perhaps hundreds of millions of rows. Specify that the right local index of the join have index type **`pi_HLEVELDB`**. This stores the dimension data in the right local index in an on-disk store. After the dimension data has been fully loaded, the fact stream can be started. Join matches are made against the dimensional events in the on-disk store, using an MRU memory cache for lookups and a filter to minimize disk seeks.

Figure 14.1 *Streaming Data into a Join Window Using Two Different Stateful Index Types*



The following C++ code fragment implements the model.

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

    dfESPcontquery *cq_01 =    project_01->newContquery("cq_01");
         dfESPstring schema_01 =
           dfESPstring("S_ID*:string,S_Plan:string,S_gid:string,S_flag:string");

    dfESPwindow_source *Dim =
           cq_01->newWindow_source((dfESPstring)"Dim",
                  dfESPindextypes::pi_EMPTY, schema_01);
           Dim->setInsertOnly();

    dfESPwindow_source *Fact = cq_01->newWindow_source((dfESPstring)"Fact",
                           dfESPindextypes::pi_EMPTY, schema_01);
        Fact->setInsertOnly();

    dfESPwindow_join *Join =
```

```
        cq_01->newWindow_join((dfESPstring)"Join", dfESPwindow_join::jt_LEFTOUTER,
                              dfESPindextypes::pi_EMPTY, false, true);
    Join->setRightIndexType(dfESPindextypes::pi_HLEVELDB);

Join->setJoinConditions("l_S_ID==r_S_ID");

Join->addNonKeyFieldCalc("r_S_ID");
Join->addNonKeyFieldCalc("l_S_Plan");
Join->addNonKeyFieldCalc("r_S_Plan");

Join->setFieldSignatures("r_S_ID:string,l_S_Plan:string,r_S_Plan:string");

cq_01->addEdge(Fact, Join);
cq_01->addEdge(Dim, Join);
```

Stateless Left Outer Join: Dimensional Updates with No-Regeneration

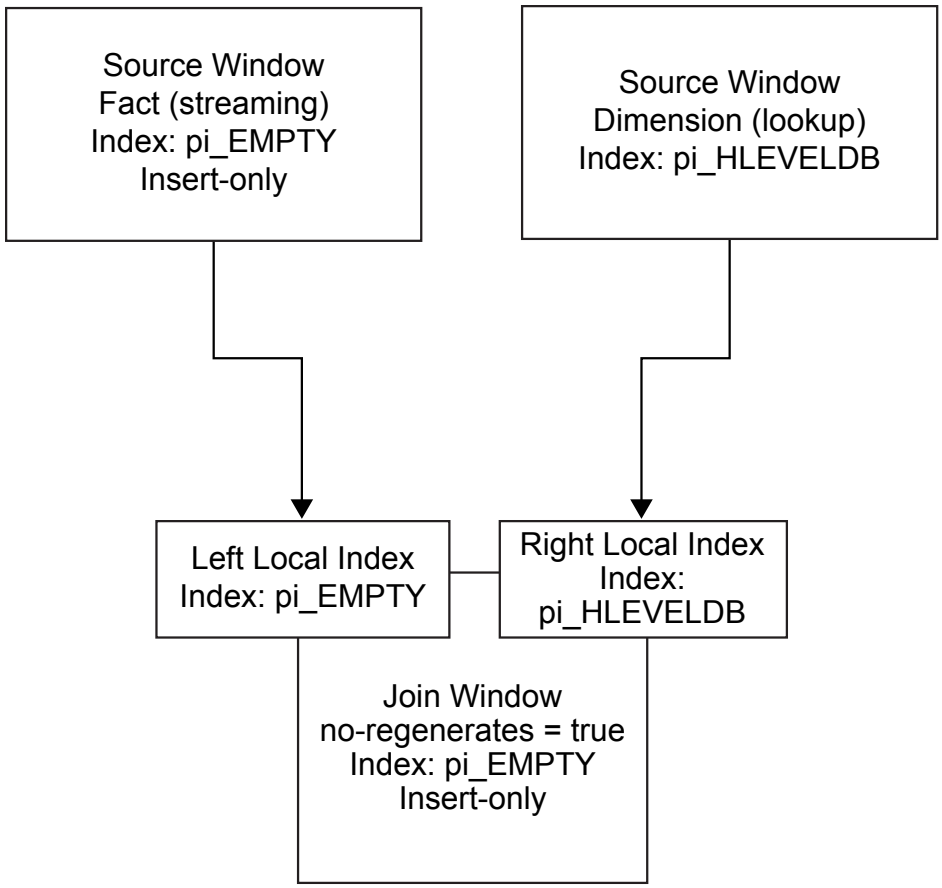
Now suppose you want to periodically update or reload the dimension table in this model. Thus, the dimensional data is no longer Insert only. To correctly resolve opcodes, the source window into which the dimensional data flows must have a stateful index.

To do this, you modify the previous model to put the dimension source data in an on-disk store. Store the following indexes to disk:

1. the primary index for the dimension source window
2. the right local index for the join

Again, the join is set to **`no-regenerates`** so that dimensional changes affect only new data. Because the **`pi_HLEVELDB`** index type does not support a retention policy, the dimension data should be naturally bounded. That is, it can encompass a very large number of events, but not an infinite number.

Figure 14.2 *Streaming Data into a Join Window When the Dimension Table Uses the pi_HLEVELDB Index Type*



The following XML code implements the model:

```
<project name='pr_01' pubsub='manual' threads='4' disk-store-path='/tmp/jones>
      <contqueries>
        <contquery name='cq_01'>
          <windows>
            <window-source name='Dim' index='pi_HLEVELDB'>
              <schema>
                <fields>
                  <field name='S_ID' type='string' key='true' />
                  <field name='S_Plan' type='string' />
                  <field name='S_gid' type='string' />
                  <field name='S_flag' type='string' />
                </fields>
              </schema>
            </window-source>
            <window-source name='Fact' index='pi_EMPTY' insert-only='true'>
              <schema>
                <fields>
                  <field name='S_ID' type='string' key='true' />
                  <field name='S_Plan' type='string' />
                  <field name='S_gid' type='string' />
                  <field name='S_flag' type='string' />
                </fields>
              </schema>
```

```
        </window-source>
        <window-join name='join_w' index='pi_EMPTY'>
          <join type='leftouter' left='Fact' right='Dim'
                right-index='pi_HLEVELDB' no-regenerates='true'>
            <conditions>
              <fields left='S_ID' right='S_ID'/>
            </conditions>
          </join>
          <output>
            <field-selection name='r_S_ID' source='r_S_ID'/>
            <field-selection name='l_S_Plan' source='l_S_Plan'/>
            <field-selection name='r_S_Plan' source='r_S_Plan'/>
          </output>
        </window-join>
      </windows>
      <edges>
        <edge source='Fact' target='join_w'/>
        <edge source='Dim' target='join_w'/>
      </edges>
    </contquery>
  </contqueries>
</project>
```

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. This index does not store events.

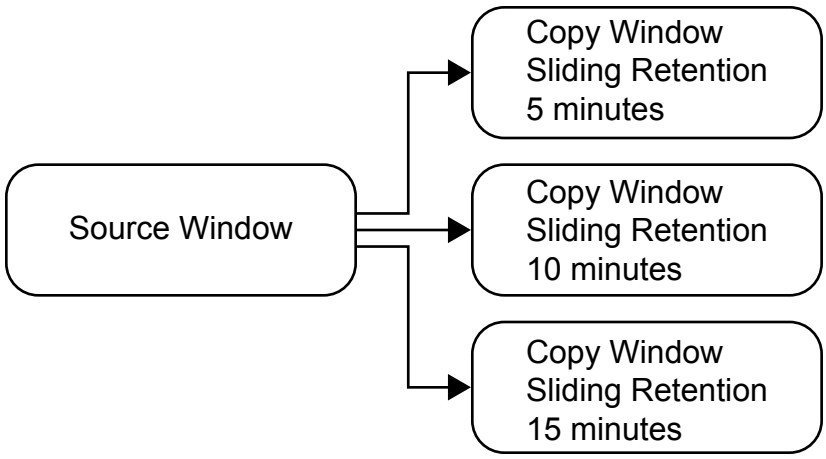
The following restrictions apply to source windows that use the empty index type.

- No restrictions apply if the source window is set to "Insert only.". For more information, see the **`setInsertOnly`** call in “dfESPwindow_source” on page 103.
- 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 14.3 *Copy Windows*



Persist and Restore Operations

SAS Event Stream Processing enables you to do the following:

- persist a complete model state to a file system
- restore a model from a persist directory that had been created by a previous persist operation
- persist and restore an entire engine
- persist and restore a project

To create a persist object for a model, provide a pathname to the class constructor: **`dfESPpersist(char *baseDir);`** The ***`baseDir`*** parameter can point to any valid directory, including disks shared among multiple running event stream processors.

After providing a pathname, call either of these two public methods:

```
bool persist();
bool restore(bool dumpOnly=false);
// dumpOnly = true means do not restore, just walk and print info
```

The **`persist()`** method can be called at any time. Be aware that it is expensive. Event block injection for all projects is suspended, all projects are quiesced, persist data is gathered and written to disk, and all projects are restored to normal running state.

The **`restore()`** method should be invoked only before any projects have been started. If the persist directory contains no persist data, the **`restore()`** call does nothing.

The persist operation is also supported by the C and Java publish/subscribe APIs. These API functions require a *host*:*port* parameter to indicate the target engine.

The C publish/subscribe API method is as follows: **`int C_dfESPpubsubPersistModel(char *hostportURL, const char *persistPath)`**

The Java publish/subscribe API method is as follows: **boolean persistModel(String hostportURL, String persistPath)**

One application of the persist and restore feature is saving state across event stream processor system maintenance. In this case, the model includes a call to the **restore()** function described previously before starting any projects. To perform maintenance at a later time on the running engine:

1. Pause all publish clients in a coordinated fashion.
2. Make one client execute the publish/subscribe persist API call described previously.
3. Bring the system down, perform maintenance, and bring the system back up.
4. Restart the event stream processor model, which executes the **restore()** function and restores all windows to the states that were persisted in step 2.
5. Resume any publishing clients that were paused in step 1.

To persist an entire engine, use the following functions:

```
bool dfESPengine::persist(const char * path);

void dfESPengine::set_restorePath(const char *path);
```

The path that you specify for **persist** can be the same as the path that you specify for **set_restorePath**.

To persist a project, use the following functions:

```
bool dfESPproject::persist(const char *path)

bool dfESPproject::restore(const char *path);
```

Start an engine and publish data into it before you persist it. It can be active and receiving data when you persist it.

To persist an engine, **call dfESPengine::persist(*path*);**. The system does the following:

1. pauses all incoming messages (suspends publish/subscribe)
2. finish processing any queued data
3. after all queued data has been processed, persist the engine state to the specified directory, creating the directory if required
4. after the engine state is persisted, resume publish/subscribe and enable new data to flow into the engine

To restore an engine, initialize it and call **dfESPengine::set_restorePath(*path*);**. After the call to **dfESPengine::startProjects()** is made, the entire engine state is restored.

To persist a project call **dfESPproject::persist(*path*);**. The call turns off publish/subscribe, quiesces the system, persists the project, and then re-enables publish/subscribe. The path specified for restore is usually the same as that for persist.

To restore the project, call **dfESPproject::restore(*path*);** before the project is started. Then call **dfESPengine::startProject(*project*);**

Gathering and Saving Latency Measurements

The **dfESPlatencyController** class supports gathering and saving latency measurements on an event stream processing model. Latencies are calculated by storing 64-bit microsecond granularity timestamps inside events that flow through windows enabled for latency measurements.

In addition, latency statistics are calculated over fixed-size aggregations of latency measurements. These measurements include average, minimum, maximum, and standard deviation. The aggregation size is a configurable parameter. You can use an instance of the latency controller to measure latencies between any source window and some downstream window that an injected event flows through.

The latency controller enables you to specify an input file of event blocks. The rate at which those events are injected into the source window. It buffers the complete input file in memory before injecting to ensure that disk reads do not skew the requested inject rate.

Specify an output text file that contains the measurement data. Each line of this text file contains statistics that pertain to latencies gathered over a bucket of events. The number of events in the bucket is the configured aggregation size. Lines contain statistics for the next bucket of events to flow through the model, and so on.

Each line of the output text file consists of three tab-separated columns. From left to right, these columns contain the following:

- the maximum latency in the bucket
- the minimum latency in the bucket
- the average latency in the bucket

You can configure the aggregation size to any value less than the total number of events. A workable value is something large enough to get meaningful averages, yet small enough to get several samples at different times during the run.

If publish/subscribe clients are involved, you can also modify publisher/subscriber code or use the file/socket adapter to include network latencies as well.

To measure latencies inside the model only:

1. Include "**int/dfESPlatencyController.h**" in your model, and add an instance of the **dfESPlatencyController** object to your **main()**.
2. Call the following methods on your **dfESPlatencyController** object to configure it:

Method	Description
void set_playbackRate(int32_t r)	Sets the requested inject rate.
void set_bucketSize(int32_t bs)	Sets the bucketSize parameter previously described.

Method	Description
`void set_maxEvents(int32_t me)`	Sets the maximum number of events to inject.
`void set_oFile(char *ofile)`	Sets the name of the output file containing latency statistics.
`void set_iFile(char *ifile)`	Sets the name of the input file containing binary event block data.

3. Add a subscriber callback to the window where you would like the events to be timestamped with an ending timestamp. Inside the callback add a call to this method on your **`dfESPlatencyController`** object: **`void record_output_events(dfESPeventblock *ob)`**. This adds the ending timestamp to all events in the event block.
4. After starting projects, call these methods on your **`dfESPlatencyController`** object:

Method	Description
`void set_injectPoint(dfESPwindow_source *s)`	Sets the source window in which you want events time stamped with a beginning timestamp.
`void read_and_buffer()`	Reads the input event blocks from the configured input file and buffers them.
`void playback_at_rate()`	Time stamps input events and injects them into the model at the configured rate, up to the configured number of events.

5. Quiesce the model and call this method on your **`dfESPlatencyController`**object: **`void generate_stats()`**. This writes the latency statistics to the configured output file.

To measure model and network latencies by modifying your publish/subscribe clients:

1. In the model, call the **`dfESPengine setLatencyMode()`** function before starting any projects.
2. In your publisher client application, immediately before calling **`C_dfESPpublisherInject()`**, call **`C_dfESPlibrary_getMicroTS()`** to get a current timestamp. Loop through all events in the event block and for each one call **`C_dfESPevent_setMeta(event, 0, timestamp)`** to write the timestamp to the event. This records the publish/subscribe inject timestamp to meta location 0.

3. The model inject and subscriber callback timestamps are recorded to meta locations 2 and 3 in all events automatically because latency mode is enabled in the engine.
4. Add code to the inject loop to implement a fixed inject rate. See the latency publish/subscribe client example for sample rate limiting code.
5. In your subscriber client application, include "**`int/dfESPlatencyController.h`**" and add an instance of the **`dfESPlatencyController`** object.
6. Configure the latency controller bucketSize and playbackRate parameters as described previously.
7. Pass your latency controller object as the context to **`C_dfESPsubscriberStart()`** so that your subscriber callback has access to the latency controller.
8. Make the subscriber callback pass the latency controller to **`C_dfESPlatencyController_recordExtraOutputEvents()`**, along with the event block. This records the publish/subscribe callback timestamp to meta location 4.
9. When the subscriber client application has received all events, you can generate statistics for latencies between any pair of the four timestamps recorded in each event. First call **`C_dfESPlatencyController_setOFile()`** to set the output file. Then write the statistics to the file by calling **`C_dfESPlatencyController_generateStats()`** and passing the latency controller and the two timestamps of interest. The list of possible timestamp pairs and their time spans are as follows:
   - (0, 2) – from inject by the publisher client to inject by the model
   - (0, 3) – from inject by the publisher client to subscriber callback by the model
   - (0, 4) – from inject by the publisher client to callback by the subscriber client (full path)
   - (2, 3) – from inject by the model to subscriber callback by the model
   - (2, 4) – from inject by the model to callback by the subscriber client
   - (3, 4) – from subscriber callback by the model to callback by the subscriber client
10. To generate further statistics for other pairs of timestamps, reset the output file and call **`C_dfESPlatencyController_generateStats()`** again.

To measure model and network latencies by using the file/socket adapter, run the publisher and subscriber adapters as normal but with these additional switches:

Publisher	
—r *rate*	Specifies the requested transmit rate in events per second.
-m *maxevents*	Specifies the maximum number of events to publish.
-p	Specifies to buffer all events prior to publishing.
-n	Enables latency mode.

Subscriber	
-r *rate*	Specifies the requested transmit rate in events per second.
-a *aggrsize*	Specifies the aggregation bucket size.
-n	Enables latency mode.

The subscriber adapter gathers all four timestamps described earlier for the windows specified in the respective publisher and subscriber adapter URLs. At the end of the run, it writes the statistics data to files in the current directory. These files are named "latency_**`transmit rate_high timestamp_low timestamp`**", where the high and low timestamps correspond to the timestamp pairs listed earlier.

Chapter 15

Using the Publish/Subscribe API

Overview to the Publish/Subscribe API **221**

Understanding Publish/Subscribe API Versioning **222**

Using the C Publish/Subscribe API **223**
The C Publish/Subscribe API from the Engine's Perspective 223
The C Publish/Subscribe API from the Client's Perspective 224
Functions for the C Publish/Subscribe API 225

Using the Java Publish/Subscribe API **237**
Overview to the Java Publish/Subscribe API 237
Using High Level Publish/Subscribe Methods 238
Using Methods That Support Google Protocol Buffers 240
Using User-supplied Callback Functions 240
Using Alternative Transport Libraries for Java Clients 241

Publish/Subscribe API Support for Google Protocol Buffers **243**
Overview to Publish/Subscribe API Support for Google Protocol Buffers 243
Converting Nested and Repeated Fields in Protocol Buffer Messages to an Event Block 244
Converting Event Blocks to Protocol Buffer Messages 245
Support for Transporting Google Protocol Buffers 245

Publish/Subscribe API Support for JSON Messaging **246**
Overview 246
Converting Nested Fields in JSON Messages to an Event Block 246
Converting Event Blocks to JSON Messages 247
Support for Transporting JSON Messages 247

Overview to the Publish/Subscribe API

SAS Event Stream Processing provides publish/subscribe application programming interfaces (APIs) for C and for Java. Use these APIs 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 APIs support TCP/IP networking. Thus, publish and subscribe applications can run on any machine with network access to an event stream processing engine, or on the

engine platform itself. APIs are available on all architectures supported by SAS Event Stream Processing Engine.

The publish/subscribe APIs handle cross-platform usage. For example, you can subscribe using the Java API to event streams in an engine that runs on Linux even though the byte order Endianness is different.

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

Connectors are in-process classes that publish and subscribe to and from event stream processing windows. For more information, see Chapter 16, “Using Connectors,” on page 249 .

Adapters are stand-alone executable files that publish and subscribe, potentially over a network. For more information, see Chapter 17, “Using Adapters,” on page 297.

Note: The APIs provide cross-platform connectivity and Endianness compatibility between the application and other networked applications, clients, and data feeds. The APIs are IPv4 compliant.

Understanding Publish/Subscribe API Versioning

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

- protocol version
- command version (the release number)

It checks to determine whether this information matches that of the server or is forward compatible with the server. Versioning enables SAS Event Stream Processing 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, if the software release is earlier than 1.2), then the following message is logged:

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

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

Using the C Publish/Subscribe API

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

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

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

if (engine == NULL) {
     cerr <<"Error: dfESPengine::initialize() failed\n";
     return 1;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

Clients that want to subscribe from or publish to an engine's 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 application or server using **C_dfESPpubsubConnect()**.

Clients that implement a publisher can then call **C_dfESPpublisherInject()** as needed to publish event blocks into the 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 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 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 application or server at any time to discover currently running windows, continuous queries, and projects in various granularities. This information is returned to the client in the form of a list of strings that represent names, which might subsequently be used to build URL strings to pass to **`C_dfESPsubscriberStart()`** or **`C_dfESPpublisherStart()`**. See the function description for a list of supported queries.

Functions for the C Publish/Subscribe API

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

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

Parameters:

`level`
the logging level

`logConfigPath`
the full pathname to the log configuration file

Return values:

1
success

0
failure — an error is written to the log

Note: This function initializes 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 written to the log).

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

clientObjPtr C_dfESPGDpublisherStart()

Parameters: Same parameters and return value as **`C_dfESPpublisherStart()`**. Additional required parameter: a Guaranteed Delivery callback function pointer of type **`C_dfESPGDpublisherCB_func`**. Additional required parameter: filename of this publisher's guaranteed delivery configuration file.

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

Parameters:

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

Specifying the client collapse requirement is optional. By default, it is false. When true, UPDATE_BLOCK events are converted to UPDATE events in order to make subscriber output publishable.

`serverURL` format
"**`dfESP://`*`host:port/project/contquery/window`*`?snapshot=true`** | **`false`**<“**`collapse=true`** | **`false`**>

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

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

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

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

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

clientObjPtr C_dfESPGDsubscriberStart()

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

int C_dfESPpubsubConnect(clientObjPtr *client*)

Parameter:

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

Return values:

1
success

0
failure — error written to the log

Note: This function attempts to establish a connection with the application or server.

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

Parameters:

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

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

Return values:

1
success

0
failure — error written to the log

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

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

Parameters:

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

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

Return values:

1
success

0
failure — error written to the log

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

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

Parameters:

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

eventBlock
the event block to inject into the engine. The block is injected into the source window, continuous query, and project associated with the passed client object.

Return values:

1
success

0
failure — error written to the log

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

C_dfESPstringV C_dfESPpubsubQueryMeta(char **queryURL*)

Parameter:

queryURL
string representing the query to be posted to the 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 written to the 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. This mechanism enables a client to discover projects, continuous queries, windows, window schema, and window edges currently running in the engine. This mechanism has no dependencies or interaction with any other activity performed by the client. The function opens an independent socket to send the query and closes the socket upon receiving the query reply.

Supported formats of *queryURL*

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

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

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

C_dfESPstringV C_dfESPpubsubGetModel(char **queryURL*)

Parameter: ***queryURL***
string representing the query to be posted to engine.

Supported formats of *queryURL* are as follows:

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

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

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

void C_dfESPpubsubShutdown()

Shutdown publish/subscribe services

int C_dfESPpubsubPersistModel(char ****hostportURL*, const char ****persistPath***)**

Parameters:

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

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

Return values:

1
success

0
failure — error written to the log

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

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

Parameters:

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

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

Return values:

1
success

0
failure — error written to the log.

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

int C_dfESPsubscriberMaxQueueSize(char *_serverURL_, int _maxSize_, int _block_)

Parameters:

serverURL
String for the server URL in one of the following forms:

- **"dfESP://*host*:*port*"**
- **"dfESP://*host*:*port*/*project*"**
- **"dfESP://*host*:*port*/*project*/*contquery*"**
- **"dfESP://*host*:*port*/*project*/*contquery*/*window*"**

maxsize
the maximum number of event blocks that can be queued for any single subscriber to a window in the *serverURL*

block
set to 1 to wait for queue size to fall below *maxSize*, else disconnect the client

Return values:

1
success

0
failure - error written to the log

Note: Use this function to configure the maximum size of all queues used to enqueue event blocks sent to subscribers in a project, query, or window. Use it to limit the amount of memory consumed by these queues. The block parameter specifies the behavior when the maximum is hit.

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Parameters:

psLib
Number representing the client/server transport

Supported values of *psLib* are as follows:

- ESP_PSLIB_NATIVE (default)
- ESP_PSLIB_SOLACE — In this mode a client configuration file named solace.cfg must be present in the current directory to provide appliance connectivity parameters.
- 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.
- ESP_PSLIB_RABBITMQ — In this mode, a client configuration filename **rabbitmq.cfg** must be present in the current directory to provide Rabbit MQ server connectivity parameters.

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Solace configuration file format:

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

Tervela configuration file format:

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

RabbitMQ configuration file format:

```
rabbitmq
{
host = "my.machine.com"
port = "5672"
exchange = "SAS"
userid = "guest"
password = "guest"
}

sas
{
buspersistence = false
queuename = "subpersist"
protobuf = false
protofile = "./GpbHistSimFactory.proto"
protomsg = "GpbTrade"
json = false
noreplay = false
}
```

Note: The **buspersistence** and **queuename** parameters mean different things for readers or writers of Rabbit MQ messages. For a writer, **queuename** is always ignored. **If buspersistence = false**, messages are sent in non-persistent delivery mode. Otherwise, delivery mode is persistent. For a reader, **queuename** is always used when creating the receive queue. If **buspersistence = false**, all queues and exchanges created by the client are non-durable and auto-delete. If **buspersistence = true**, all exchanges and queues are durable and not auto-delete. The **noreplay** parameter is false by default. When set to true, received messages are acknowledged even when **buspersistence** is enabled.

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Return values:

1
success

0
failure

Note: This function call is optional, but if called it must be called before calling **`C_dfESPpubsubInit()`**. It modifies the transport used between the client and the 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, ESP_PSLIB_TERVELA, or ESP_PSLIB_RABBITMQ to indicate that the client's TCP/IP peer is a Solace appliance, a Tervela appliance, or a Rabbit MQ server. This mode requires that the engine runs a Solace, Tervela, or Rabbit MQ 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: When using the Solace, Tervela, or Rabbit MQ transports, the following publish/subscribe API functions are not supported:

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

C_dfESPGDsubscriberAck(clientObjPtr *client*, CdfESPeventblock *eventblock*)

Parameters:

- *client* pointer to a client object returned by **`C_dfESPGDsubscriberStart()`**
- *eventBlock* pointer to the event block being acknowledged back to the publisher. The event block must not be freed before this function returns.

Return values:

- 1 = success
- 0 = failure

C_dfESPGDpublisherCB_func()

Parameters: Signature of the guaranteed delivery publisher callback function passed to **`C_dfESPGDpublisherStart()`**. This function is invoked by the API to return the guaranteed delivery status of a published event block back to the publisher.

- 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:

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

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

Return values:

1
success

0
failure

Note: This function call is optional, but if called it must be called after **C_dfESPsubscriberStart()**, **C_dfESPpublisherStart()**, **C_dfESPGDsubscriberStart()**, or **C_dfESPGDpublisherStart()** and before **C_dfESPpubsubConnect()**. It modifies the size of the buffers used for socket Read and Write operations. By default this size is 16MB

protobuffObjPtr C_dfESPpubsubInitProtobuff(char **protoFile*, char **msgName*, C_dfESPschema *C_schema*, char **dateFormat*)

Parameters:

protoFile
Path to the Google .proto file that contains the message definition.

msgName
Name of the target message definition in the Google .proto file.

C_schema
Pointer to the window schema.

dateFormat
Date format for CSV conversions. Set to null for the default format.

Return value: A pointer to a **protobuff** object, which is passed to all other **protobuff** API functions. NULL when there is a failure.

C_dfESPeventblock C_dfESPprotobuffToEb(protobuffObjPtr *protobuff*, void **serializedProtobuff*)

Parameters:

protobuff
pointer to the object created by **C_dfESPpubsubInitProtobuff()**

serializedProtobuff
pointer to serialized **protobuff** received on a socket

C_dfESPeventblock C_dfESPprotobuffToEb(protobuffObjPtr *protobuff*, void **serializedProtobuff*)

Return value: Event block pointer

void *C_dfESPebToProtobuff(protobuffObjPtr *protobuff*, C_dfESPeventblock *C_eb*, int32_t *index*)

Parameters:

protobuff
pointer to the object created by `C_dfESPpubsubInitProtobuff()`

C_eb
event block pointer

index
index of the event to convert in the event block

Return value: pointer to serialized `protobuff`

void C_dfESPdestroyProtobuff(protobuffObjPtr *protobuff*, void **serializedProtobuff*)

Parameters:

protobuff
pointer to the object created by `C_dfESPpubsubInitProtobuff()`

serializedProtobuff
pointer to serialized `protobuff` received on a socket

jsonObjPtr C_dfESPpubsubInitJson(C_dfESPschema *C_schema*, char **dateFormat*)

Parameters:

C_schema
pointer to the window schema

dateFormat
Date format for CSV conversions. Set to null for the default format.

Return value: A pointer to a JSON object, which is passed to all other JSON API functions. NULL when there is a failure.

C_dfESPeventblock C_dfESPjsonToEb(jsonObjPtr *json*, void **serializedJson*, int32_t *maxEvents*)

Parameters:

json
pointer to the object created by `C_dfESPpubsubInitJson()`

serializedJson
pointer to a serialized JSON received on a socket

maxEvents
limit the number of events created to this value. A value of 0 means no limit.

Return value: event block pointer

void *C_dfESPebToJson(jsonObjPtr *json*, C_dfESPeventblock *C_eb*)	
Parameters:	***json*** pointer to the object created by **C_dfESPpubsubInitJson()** ***C_eb*** event block pointer
Return value:	pointer to a serialized JSON

A C library provides a set of functions to enable client developers to analyze and manipulate the event stream processing objects from the 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

SAS Event Stream Processing 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:

- **com.sas.esp.api.pubsub**
  - **com.sas.esp.api.pubsub.clientHandler**

    - **com.sas.esp.api.pubsub.clientCallbacks**
- **com.sas.esp.api.server**
    - **com.sas.esp.api.server.datavar**
    - **com.sas.esp.api.server.event**
    - **com.sas.esp.api.server.eventblock**
    - **com.sas.esp.api.server.library**
    - **com.sas.esp.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 parameters and usage for the Java publish/subscribe API are the same as for the equivalent calls for the C publish/subscribe API.

The C API references and Java interface references are available at **$DFESP_HOME/doc/html**.

Using High Level Publish/Subscribe Methods

The following high-level publish/subscribe methods are defined in the following interface reference: **com.sas.esp.api.pubsub.clientHandler**.

Method	Description
boolean init(Level *level*)	Initialize publish/subscribe services
dfESPclient publisherStart(String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*)	Start a publisher.
dfESPclient subscriberStart(String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*)	Start a subscriber
boolean connect(dfESPclient *client*)	Connect to the Event Stream Processor application or server
boolean publisherInject((dfESPclient *client*, dfESPeventblock eventblock)	Publish event blocks
ArrayList< String > *queryMeta* (String *queryURL*)	Query model metadata
ArrayList< String > getModel(String *queryURL*)	Query model windows and their edges
boolean disconnect (dfESPclient *client*, boolean *block*)	Disconnect from the event stream processor

Method	Description
boolean stop (dfESPclient *client*, boolean *block*)	Stop a subscriber or publisher
void shutdown ()	Shutdown publish/subscribe services
boolean setBufferSize(dfESPclient *client*, int *mbytes*)	Change the default socket read and write buffer size
dfESPclient GDsubscriberStart (String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*, String *configFile*)	Start a guaranteed delivery subscriber
dfESPclient GDpublisherStart (String *serverURL*, clientCallbacks *userCallbacks*, Object *ctx*, String *configFile*)	Start a guaranteed delivery publisher
long GDpublisherGetID()	Get a sequentially unique ID to write to an event block to be published using guaranteed delivery
boolean GDsubscriberAck(dfESPclient *client*, dfESPeventblock *eventblock*)	Trigger a guaranteed delivery acknowledgment
boolean persistModel(String *hostportURL*, String *persistPath*)	Instruct a running engine to persist its current state to disk
boolean quiesceProject(String *projectURL*, dfESPclient *client*)	Instruct a running engine to quiesce a specific project in the model.
boolean subscriberMaxQueueSize(String *serverURL*, int *maxSize*, boolean *block*)	Configure the maximum size of the queues in the event stream processing server that are used to enqueue event blocks sent to subscribers. Set **block** to 1 to wait for queue size to fall below**maxSize**, else disconnect the client.

For more information, see**$DFESP_HOME/doc/html/index.html**. Search the **Classes** page for **client handler**.

Using Methods That Support Google Protocol Buffers

The following methods support Google Protocol Buffers. They are defined in this interface reference: **com.sas.esp.api.pubsub.protobufInterface**.

Method	Description
boolean init(String *fileDescriptorSet*, String *msgName*, dfESPschema *schema*)	Initialize the library that supports Google Protocol Buffers.
dfESPeventblock protobufToEb(byte[] *serializedProtobuf*)	Convert a **protobuf** message to an event block.
byte[] ebToProtobuf(dfESPeventblock *eb*, int *index*)	Convert an event in an event block to a **protobuf** message.

For more information, see "Publish/Subscribe API Support for Google Protocol Buffers" on page 243.

Using User-supplied Callback Functions

The **com.sas.esp.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 com.sas.esp.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 com.sas.esp.api.pubsub.clientCallbacks.dfESPpubsubErrorCB_func
     (clientFailures failure, clientFailureCodes code, Object ctx)
```

clientFailures and client FailureCodes are defined in interface references **com.sas.esp.api.pubsub.clientFailures** and **com.sas.esp.api.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 com.sas.esp.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

Using Alternative Transport Libraries for Java Clients

Alternative transport libraries enable a Java publish/subscribe client application to send and receive event blocks through a mechanism other than a direct TCP/IP connection to the client:

- Rabbit MQ Java libraries enable sending and receiving through the Rabbit MQ server.
- Solace Java libraries enable sending and receiving through the Solace appliance.
- Tervela Java libraries enable sending and receiving through the Tervela appliance.

These libraries enable the Java equivalent of the C publish/subscribe API method, substituting a Solace transport or a Tervela transport. When the engine is configured for 1 + N-Way Failover using Rabbit MQ, Solace, or Tervela, Java publish/subscribe clients must use the corresponding client library to guarantee successful failover.

To substitute one of these libraries in your Java publish/subscribe client application, insert the corresponding JAR filename in front of **dfx-esp-api.jar** in your classpath, as shown here:

- For Rabbit MQ, the JAR file is **dfx-esp-rabbitmq-api.jar**.
- For Solace, the JAR file is **dfx-esp-solace-api.jar**.
- For Tervela, the JAR file is **dfx-esp-tervela-api.jar**.

If you are using the Rabbit MQ library, you must also install the native Rabbit MQ Java client libraries (**rabbitmq-client.jar**) on your system. Obtain them at http://www.rabbitmq.com/java-client.html. Then add **rabbitmq-client.jar** to your classpath.

Your current working directory must also include a corresponding configuration file as shown here:

- For RabbitMQ this file must be named **rabbitmq.cfg**.
- For Solace, it must be named **solace.cfg**.
- For Tervela, it must be named **client.config**.

Here is a sample configuration file for RabbitMQ:

```
{
rabbitmq =
{
host = "my.machine.com";
port = "5672";
exchange = "SAS";
userid = "guest";
password = "guest";
}
sas =
{
buspersistence = false;
queuename = "subpersist";
protobuf = false;
protofile = "./GpbHistSimFactory.proto";
protomsg = "GpbTrade";
noreplay = false;
}
}
```

The **buspersistence** and **queuename** parameters mean different things for publishers and subscribers.

- For a publisher, **queuename** is always ignored. If **buspersistence = false**, messages are sent in non-persistent delivery mode. Otherwise, delivery mode is persistent.
- For a subscriber, **buspersistence = false** means that all queues and exchanges created by the client are non-durable and auto-delete and that the **queuename** parameter is ignored. If **buspersistence = true**, all exchanges and queues are durable and not auto-delete and the **queuename** in the durable receive queue is fixed.

The **noreplay** parameter is false by default. When set to true, received messages are acknowledged even when **buspersistence** is enabled.

Here is a sample configuration file for Solace:

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

Here is a sample configuration file for Tervela:

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

Publish/Subscribe API Support for Google Protocol Buffers

Overview to Publish/Subscribe API Support for Google Protocol Buffers

SAS Event Stream Processing provides a library to support Google Protocol Buffers. This library provides conversion methods between an event block in binary format and a serialized Google protocol buffer (**protobuf**).

To exchange a **protobuf** with an event stream processing server, a publish/subscribe client using the standard publish/subscribe API can call **C_dfESPpubsubInitProtobuff()** to load the library that supports Google Protocol Buffers. Then a publisher client with source data in **protobuf** format can call **C_dfESPprotobuffToEb()** to create event blocks in binary format before calling **C_dfESPpublisherInject()**. Similarly, a subscriber client can convert a received events block to a **protobuf** by calling **C_dfESPebToProtobuff()** before passing it to a **protobuf** handler.

Note: The server side of an event stream processing publish/subscribe connection does not support Google Protocol Buffers. It continues to send and receive event blocks in binary format only.

The SAS Event Stream Processing Java publish/subscribe API contains a **protobuf** JAR file that implements equivalent methods for Java publish/subscribe clients. In order to load the library that supports Google Protocol Buffers, you must have installed the standard Google Protocol Buffers run-time library. The SAS Event Stream Processing run-time environment must be able to find this library. For Java, you must have installed the Google **protobuf** JAR file and have included it in the run-time class path.

A publish/subscribe client connection exchanges events with a single window using that window's schema. Correspondingly, a **protobuf** enabled client connection uses a single fixed **protobuf** message type, as defined in a message block in a **.proto** file. The library that supports Google Protocol buffers dynamically parses the message definition, so no precompiled message-specific classes are required. However, the Java library uses a **.desc** file instead of a **.proto** file, which requires you to run the Google **protoc** compiler on the **.proto** file in order to generate a corresponding **.desc** file.

For C clients, the name of the **.proto** file and the enclosed message are both passed to the library that supports Google Protocol Buffers in the **C_dfESPpubsubInitProtobuff()** call. This call returns a **protobuf** object instance, which is then passed in all subsequent **protobuf** calls by the client. This instance is specific to the **protobuf** message definition. Thus, it is valid as long as the client connection to a specific window is up. When the client stops and restarts, it must obtain a new **protobuf** object instance.

For Java clients, the process is slightly different. The client creates an instance of a **dfESPprotobuf** object and then calls its **init()** method. Subsequent **protobuf** calls are made using this object's methods, subject to the same validity scope described for the C++ **protobuf** object.

Conversion between a binary event block and a **protobuf** is accomplished by matching fields in the **protobuf** message definition to fields in the schema of the associated publish/subscribe window. Ensure that the **protobuf** message definition and the window schema are compatible. When the **protobuf** message definition contains optional fields, ensure that they are included in the window schema. If a received **protobuf** message is missing an optional field, the corresponding field in the event is set to null. Conversely, when building a **protobuf** and a field of an event contains null, the corresponding **protobuf** field is left unset, and therefore must be defined as optional in the **.proto** file.

The following mapping of event stream processor to Google Protocol Buffer data types are supported:

Event Stream Processor Data Type	Google Protocol Buffer Data Type
ESP_DOUBLE	TYPE_DOUBLE TYPE_FLOAT
ESP_INT64	TYPE_INT64 TYPE_UINT64 TYPE_FIXED64 TYPE_SFIXED64 TYPE_SINT64
ESP_INT32	TYPE_INT32 TYPE_UINT32 TYPE_FIXED32 TYPE_SFIXED32 TYPE_SINT32 TYPE_ENUM
ESP_UTF8STR	TYPE_STRING
ESP_DATETIME	TYPE_INT64
ESP_TIMESTAMP	TYPE_INT64
Other mappings are currently unsupported.	

Converting Nested and Repeated Fields in Protocol Buffer Messages to an Event Block

Provided that they are supported, you can repeat the message fields of a **protobuf**. A message field of TYPE_MESSAGE can be nested, and possibly repeated as well. All of

these cases are supported when converting a **protobuf** message to an event block, observing the following policies:

- A **protobuf** message that contains nested messages requires that the corresponding schema be a flattened representation of the **protobuf** message. For example, a **protobuf** message that contains three fields where the first is a nested message with four fields, the second is not nested, and the third is a nested message with two fields requires a schema with 4 + 1 + 2 = 7 fields. Nesting depth is unbounded.
- A single **protobuf** message is always converted to an event block that contains a single event, provided that no nested message field is repeated.
- When a **protobuf** message has a non-message type field that is repeated, all the elements in that field are gathered into a single comma-separated string field in the event. For this reason, any schema field that corresponds to a repeated field in a **protobuf** message must have type ESP_UTF8STR, regardless of the field type in the **protobuf** message.
- A **protobuf** message that contains nested message fields that are repeated is always converted to an event block that contains multiple events. There is one event for each element in every nested message field that is repeated.

Converting Event Blocks to Protocol Buffer Messages

Converting an event block to a **protobuf** is conceptually similar to converting nested and repeated fields in a **protobuf** to an event block, but the process requires more code. Every event in an event block is converted to a separate **protobuf** message. For this reason, the **C_dfESPebToProtobuff()** library call takes an index parameter that indicates which event in the event block to convert. The library must be called in a loop for every event in the event block.

The conversion correctly loads any nested fields in the resulting **protobuf** message. Any repeated fields in the resulting **protobuf** message contain exactly one element, because event blocks do not support repeated fields.

Note: Event block to **protobuf** conversions support only events with the Insert opcode, because event opcodes are not copied to **protobuf** messages. Conversions of **protobuf** to event blocks always use the Upsert opcode.

Support for Transporting Google Protocol Buffers

Support for Google Protocol Buffers is available when you use the connectors and adapters that are associated with the following message buses:

- IBM WebSphere MQ
- Rabbit MQ
- Solace
- Tervela
- Tibco/RV

These connectors and adapters support transport of a **protobuf** through the message bus, instead of binary event blocks. This allows a third-party publisher or subscriber to connect to the message bus and exchange a **protobuf** with an engine without using the publish/subscribe API. The **protobuf** message format and window schema must be compatible.

The connector or adapter requires configuration of the **.proto** file and message name through the **protofile** and **protomsg** parameters. The connector converts a **protobuf** to and from an event block using the SAS Event Stream Processing library that supports Google Protocol Buffers. In addition, the C and Java Solace publish/subscribe clients also support Google Protocol Buffers when configured to do so in the **solace.cfg** client configuration file. Similarly, C RabbitMQ publish/subscribe clients support Google Protocol Buffers when they are configured to do so in the **rabbitmq.cfg** client configuration file. A **protobuf** enabled client publisher converts an event block to a **protobuf** to transport through the message bus to a third-party consumer of Google Protocol Buffers. Likewise, a **protobuf**-enabled client subscriber receives a **protobuf** from the message bus and converts it to an event block.

Publish/Subscribe API Support for JSON Messaging

Overview

SAS Event Stream Processing provides a C library to support JSON messaging. This library provides conversion methods between an event block in binary format and a serialized JSON message.

To exchange a JSON message with an event stream processing server, a publish/subscribe client that uses the standard publish/subscribe API can call **C_dfESPpubsubInitJson()** to load the library that supports JSON. Then a publisher client with source data in JSON format can call **C_dfESPprotobuffToJson()** to create event blocks in binary format. It can then call **C_dfESPpublisherInject()**.

Similarly, a subscriber client can convert a received events block to a JSON message by calling **C_dfESPebToJson()** before passing it to a JSON message handler.

Note: The server side of an event stream processing publish/subscribe connection does not support JSON messages. It continues to send and receive event blocks in binary format only.

A publish/subscribe client connection exchanges events with a single window using that window's schema. Correspondingly, a JSON-enabled client connection exchanges messages with a fixed JSON schema. However, there is no static definition of this schema. A schema mismatch between a JSON message and the related window schema is detected only at run time.

The **C_dfESPpubsubInitProtobuff()** call returns a JSON object instance, which is then passed in all subsequent JSON calls by the client. This object instance is valid only while the client connection to a specific window is up. When the client stops and restarts, it must obtain a new JSON object instance.

Fundamentally, a single JSON message maps to a single event. However, a JSON message can contain multiple events when you enclose them within a JSON array.

Converting Nested Fields in JSON Messages to an Event Block

The window schema must be a flattened representation of the JSON event schema, where window field names are a concatenation of any nested JSON tag names, separated by underscores.

Within a JSON event schema, unlimited nesting of arrays and objects is supported.

When a JSON event contains an array field, all the elements in that array are gathered into a single comma-separated string field in the event. For this reason, any schema field that corresponds to an array field in a JSON event must have type **ESP_UTF8STR**, regardless of the field type within the JSON event schema.

Integer support in JSON field values is limited to 32 bits. 64–bit integers are not supported.

The event built from a JSON event always has the Upsert opcode. The exception is when the JSON event contains a field named **opcode**.

Table 15.1 *Valid Values for the Opcode Field in a JSON Event*

Value	Opcode
i **I**	Insert
u **U**	Update
d **D**	Delete
p **P**	Upsert
s **S**	Safe delete

Converting Event Blocks to JSON Messages

All JSON events that are created from an event contain an **opcode** field. Valid values are listed in Table 15.1 on page 247.

Because the input is an event block, the resulting JSON message always has an array as its root object. Each array entry represents a single event.

Data variable type **ESP_MONEY** is not supported.

Data variable types **ESP_TIMESTAMP** and **ESP_DATETIME** are converted to a JSON string that contains the CSV representation of the field value.

Support for Transporting JSON Messages

Support for JSON messaging is available when you use the connectors and adapters associated with the following message buses:

- IBM WebSphere MQ
- RabbitMQ
- Solace

- Tervela
- Tibco/RV

These connectors and adapters support transport of JSON encoded messages through the message bus, instead of binary event blocks. This enables a third-party publisher or subscriber to connect to the message bus and exchange JSON messages with an engine without using the publish/subscribe API.

No message-format configuration is required. If the JSON schema and window schema are incompatible, a publisher that converts JSON to event blocks fails when an event block is injected. The connector converts JSON to and from an event block using the SAS Event Stream Processing library that supports JSON conversion.

C RabbitMQ publish/subscribe clients support JSON when configured to do so in the **`rabbitmq.cfg`** client configuration file. A JSON-enabled client publisher converts an event block to a JSON message to transport through the message bus to a third-party consumer of JSON messages. Likewise, a JSON enabled client subscriber receives a JSON message from the message bus and converts it to an event block.

Chapter 16

Using Connectors

Overview to Using Connectors **250**
What Do Connectors Do? 250
Connector Examples 250
Obtaining Connectors 251
Activating Optional Plug-ins 251
Setting Configuration Parameters 252
Setting Configuration Parameters in a File 252
Orchestrating Connectors 252

Using the Database Connector **254**
Overview to Using the Database Connector 254
Subscriber Event Stream Processor to SQL Data Type Mappings 257
Publisher SQL to Event Stream Processor Data Type Mappings 258
Connectivity to Netezza Databases 259

Using File and Socket Connectors **259**
Overview to File and Socket Connectors 259
XML File and Socket Connector Data Format 262
JSON File and Socket Connector Data Format 263
Syslog File and Socket Connector Notes 264
HDAT Subscribe Socket Connector Notes 264

Using the IBM WebSphere MQ Connector **265**

Using the PI Connector **267**

Using the Project Publish Connector **271**

Using the Rabbit MQ Connector **271**

Using the SMTP Subscribe Connector **276**

Using the Sniffer Connector **277**

Using the Solace Systems Connector **280**

Using the Teradata Connector **284**

Using the Tervela Data Fabric Connector **286**

Using the Tibco Rendezvous (RV) Connector **290**

Writing and Integrating a Custom Connector **293**
Writing a Custom Connector 293
Integrating a Custom Connector 295

Overview to Using Connectors

What Do Connectors Do?

Connectors use the SAS Event Stream Processing publish/subscribe API to do one of the following:

- publish event streams into source windows. Publish operations do the following, usually continuously:
  - read event data from a specified source
  - inject that event data into a specific source window of a running event stream processor
- subscribe to window event streams. Subscribe operations write output events from a window of a running event stream processor to the specified target (usually continuously).

Connectors do not simultaneously publish and subscribe.

You can find connectors in libraries located at **$DFESP_HOME/lib/plugins**. On Microsoft Windows platforms, you can find them at **%DFESP_HOME%\bin\plugins**.

All connector classes are derived from a base connector class that is included in a connector library. The base connector library includes a connector manager that is responsible for loading connectors during initialization. This library is located in **$DFESP_HOME/lib/libdfxesp_connectors-Maj.Min**, where **Maj.Min** indicates the release number for the distribution.

Connector Examples

Connector examples are available in **$DFESP_HOME/examples/cxx**. The **sample_connector** directory includes source code for a user-defined connector derived from the **dfESPconnector** base class. It also includes sample code that invokes a sample connector. For more information about how to write a connector and getting it loaded by the connector manager, see "Writing and Integrating a Custom Connector" on page 293.

The remaining connector examples implement application code that invokes existing connectors. These connectors are loaded by the connector manager at initialization. Those examples are as follows:

- **db_connector_publisher**
- **db_connector_subscriber**
- **json_connector_publisher**
- **json_connector_subscriber**
- **socket_connector_publisher**
- **socket_connector_subscriber**
- **xml_connector_publisher**
- **xml_connector_subscriber**

Obtaining Connectors

To obtain a new instance of a connector, call the **dfESPwindow::getConnector()** method. Pass the connector type as the first parameter, and pass a connector instance name as an optional second parameter

```
dfESPConnector *inputConn =
static_cast<dfESPCConnector *>(input->getConnector("fs","inputConn"));
dfESPConnector *inputConn=...
```

The packaged connector types are as follows:

- **db**
- **fs**
- **mq**
- **project**
- **rmq**
- **sniffer**
- **smtp**
- **sol**
- **tdata**
- **tva**
- **tibrv**
- **pi**

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.

Activating Optional Plug-ins

The **$DFESP_HOME/lib/plugins** directory contains the complete set of plug-in objects supported by SAS Event Stream Processing. Plug-ins that contain **"_cpi"** in their filename are connectors.

The connectors.excluded file in the **$DFESP_HOME/etc** 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. This prevents those connectors from being automatically loaded and generating errors due to missing dependencies.

You can edit connectors.excluded as needed. You can list any of the valid connector types in connectors.excluded.

Setting Configuration Parameters

Use the **setParameter()** method to set required and optional parameters for a connector. You can use **setParameter()** as many times as you need. You must set a connector's parameters before starting it.

The **type** parameter is required and is common to all connectors. It must be set to **pub** or **sub**. What additional connector configuration parameters are required depends on the connector type.

Setting Configuration Parameters in a File

You can completely or partially set configuration parameters in a configuration file. You specify a set of parameters and give that set a section label. You then can use **setParameter()** to set the **configfilesection** parameter equal to the section label. This configures the entire set. If any parameters are redundant, a parameter value that you configure separately using **setParameter()** takes precedence.

When you configure a set of parameters, the connector finds the section label in **/etc/connectors.config** and configures the parameters listed in that section.

The following lines specify a set of connector parameters to configure and labels the set TestConfig

```
[testconfig]
type=pub
host=localhost
port=33340
project=sub_project
continuousquery=subscribeServer
window=tradesWindow
fstype=binary
fsname=./sorted_trades1M_256perblock.bin
```

You can list as many parameters as you want in a section so labeled.

Orchestrating Connectors

By default, all connectors start automatically when their associated project starts and they run concurrently. Connector orchestration enables you to define the order in which connectors within a project execute, depending on the state of the connector. You can thereby create self-contained projects that orchestrate all of their inputs. Connector orchestration can be useful to load reference data, inject bulk data into a window before injecting streaming data, or with join windows.

You can represent connector orchestration as a directed graph, similar to how you represent a continuous query. In this case, the nodes of the graph are connector groups, and the edges indicate the order in which groups execute.

Connectors ordinarily are in one of three states: stopped, running, or finished. Subscriber connectors and publisher connectors that are able to publish indefinitely (for example, from a message bus) never reach finished state.

In order for connector execution to be dependent on the state of another connector, both connectors must be defined in different connector groups. Groups can contain multiple connectors, and all dependencies are defined in terms of the group, not the individual connectors.

When you add a connector to a group, you must specify a corresponding connector state as well. This state defines the target state for that connector within the group. When all connectors in a group reach their target state, all other groups dependent on that group are satisfied. When a group becomes satisfied, all connectors within that group enter running state.

Consider the following configuration that consists of four groups: G1, G2, G3, and G4:

```
G1: {<connector_pub_A, FINISHED>, <connector_sub_B, RUNNING>}
G2: {<connector_pub_C, FINISHED>}
G3: {<connector_pub_D, RUNNING>}
G4: {<connector_sub_E, RUNNING>}
```

And then consider the following orchestration:

- G1 -> G3: start the connectors in G3 after all of the connectors in G1 reach their target states.
- G2 -> G3: start the connectors in G3 after all of the connectors in G2 reach their target states. Given the previous orchestration, this means that G3 does not start until the connectors in G1 and in G2 reach their target states.
- G2 -> G4: start the connectors in G4 after all of the connectors in G2 reach their target states.

Because G1 and G2 do not have dependencies on other groups, all of the connectors in those groups start right away. The configuration results in the following orchestration:

1. When the project is started, **`connector_pub_A`**, **`connector_sub_B`**, and **`connector_pub_C`** start immediately.
2. When **`connector_pub_C`** finishes, **`connector_sub_E`** is started.
3. **`connector_pub_D`** only starts after all conditions for G3 are met, that is, when conditions for G1 and G2 are satisfied. Thus, it starts only when **`connector_pub_A`** is finished, **`connector_sub_B`** is running, and **`connector_pub_C`** is finished.

A connector group is defined by calling the project **`newConnectorGroup()`** method, which returns a pointer to a new **`dfESPconnectorGroup`** instance. If you pass only the group name, the group is not dependent on any other group. Conversely, you can also pass a vector or variable list of group instance pointers. This defines the list of groups that must all become satisfied in order for the new group to run.

After you define a group, you can add connectors to it by calling the **`dfESPconnectorGroup::addConnector()`** method. This takes a connector instance (returned from **`dfESPwindow::getConnector()`**), and its target state.

The C++ code to define this orchestration is as follows:

```
dfESPconnectorGroup*G1= project->newConnectorGroup("G1");
G1-> addConnector(pub_A, dfESPabsConnector::state_FINISHED);
G1-> addConnector(sub_B, dfESPabsConnector::state_RUNNING);

dfESPconnectorGroup*G2= project->newConnectorGroup("G2");
G2-> addConnector(pub_C, dfESPabsConnector::state_FINISHED);

dfESPConnectorGroup*G3= project->newConnectorGroup("G3",2,G1,G2);
G3-> addConnector(pub_D, dfESPabsConnector::state_RUNNING);

dfESPConnectorGroup*G4= project->newConnectorGroup("G4",1,G2);
G4-> addConnector(sub_E, dfESPabsconnector::state_RUNNING);
```

The XML code is as follows:

```
<project-connectors>
        <connector-groups>
          <connector-group name='G1'>
            <connector-entry
              connector='contQuery_name/window_for_pub_A/pb_A'
              state='finished'/>
            <connector-entry
              connector='contQuery_name/window_for_sub_B/sub_B'
              state='running'/>
          </connector-group>
          <connector-group name='G2'>
            <connector-entry
              connector='contQuery_name/window_for_pub_C/pub_C'
              state='finished'/>
          </connector-group>
          <connector-group name='G3'>
            <connector-entry
               connector='contQuery_name/window_for_pub_D/pub_D'
               state='running'/>
          </connector-group>
          <connector-group name='G4'>
            <connector-entry
               connector='contQuery_name/window_for_sub_E/sub_E'
               state='running'/>
          </connector-group>
        </connector-groups>
        <edges>
          <edge source='G1' target='G3'/>
          <edge source='G2' target='G3'/>
          <edge source='G2' target='G4'/>
        </edges>
      </project-connectors>
```

Using the Database Connector

Overview to Using the Database Connector

The database connector provided by SAS Event Stream Processing supports both publish and subscribe operations. It uses the DataDirect ODBC driver. Currently, it is certified for the following databases:

- Oracle
- MySQL
- IBM DB2
- Greenplum
- PostgreSQL
- SAP Sybase ASE
- Teradata

- Microsoft SQL Server
- IBM Informix
- Sybase IQ

The connector requires that database connectivity be available through a system Data Source Name (DSN). This DSN and the associated database user credentials are required configuration parameters for the connector.

For SAS Event Stream Processing 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 **`$DFESP_HOME/bin/dfdbconf`**. Select a driver from the list of available drivers and set the appropriate parameters for that driver. The new DSN is added to the odbc.ini file. In the odbc.ini file, replace **`$DFESP_HOME`** with the full path for the driver.
- **`dfdbview`** - This interactive tool enables the user to manually connect to the database and perform operations using SQL commands.

For Windows installations, ensure that the optional ODBC component is installed. 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 16.1 *Required Parameters for Subscriber Database Connectors*

Parameter	Description
`type`	Specifies to subscribe.

Parameter	Description
connectstring	Specifies the database DSN and user credentials in the format "**DSN=*dsn*;uid=*userid*;pwd=*password*;**"
tablename	Specifies the target table name.
snapshot	Specifies whether to send snapshot data.

Table 16.2 *Required Parameters for Publisher Database Connectors*

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

Table 16.3 *Optional Parameters for Subscriber Database Connectors*

Parameter	Description
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
commitrows	Specifies the minimum number of output rows to buffer.
commitsecs	Specifies the maximum number of seconds to hold onto an incomplete commit buffer.
ignoresqlerrors	Enables the connector to continue to write Inserts, Updates, and Deletes to the database table despite an error in a previous Insert, Update, or Delete.

Table 16.4 *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.

Parameter	Description
`transactional`	Sets the event block type to transactional. The default value is normal.
`configfilesection`	Specifies the name of the section in **`/etc/connectors.config`** to parse for configuration parameters. Specify the value as **`[`*`configfilesection`*`]`**.

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
ESP_MONEY	SQL_DOUBLE (converted to ESP_DOUBLE), SQL_FLOAT, SQL_REAL, SQL_NUMERIC (converted to SQL_NUMERIC), SQL_DECIMAL (converted to SQL_NUMERIC)
ESP_DATETIME	SQL_TYPE_DATE (only sets year/month/day), SQL_TYPE_TIME (only sets/hours/minutes/seconds), SQL_TYPE_TIMESTAMP (sets fractional seconds = 0)
ESP_TIMESTAMP	SQL_TYPE_TIMESTAMP

The following SQL mappings are not supported: SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY, SQL_BIT, SQL_TINYINT, SQL_SMALLINT

Publisher SQL to Event Stream Processor Data Type Mappings

The SQL to event stream processor data type mappings are as follows:

Publisher SQL Data Type	Event Stream Processor Data Types
SQL_CHAR	ESP_UTF8STR
SQL_VARCHAR	ESP_UTF8STR
SQL_LONGVARCHAR	ESP_UTF8STR
SQL_WCHAR	ESP_UTF8STR
SQL_WVARCHAR	ESP_UTF8STR
SQL_WLONGVARCHAR	ESP_UTF8STR
SQL_BIT	ESP_INT32, ESP_INT64
SQL_TINYINT	ESP_INT32, ESP_INT64
SQL_SMALLINT	ESP_INT32, ESP_INT64
SQL_INTEGER	ESP_INT32, ESP_INT64
SQL_BIGINT	ESP_INT64
SQL_DOUBLE	ESP_DOUBLE, ESP_MONEY (upcast from ESP_DOUBLE)
SQL_FLOAT	ESP_DOUBLE, ESP_MONEY (upcast from ESP_DOUBLE)
SQL_REAL	ESP_DOUBLE
SQL_TYPE_DATE	ESP_DATETIME (sets only year/month/day)
SQL_TYPE_TIME	ESP_DATETIME (sets only hours/minutes/seconds)
SQL_TYPE_TIMESTAMP	ESP_TIMESTAMP, ESP_DATETIME
SQL_DECIMAL	ESP_INT32 (only if scale = 0, and precision must be <= 10), ESP_INT64 (only if scale = 0, and precision must be <= 20), ESP_DOUBLE
SQL_NUMERIC	ESP_INT32 (only if scale = 0, and precision must be <= 10), ESP_INT64 (only if scale = 0, and precision must be <= 20), ESP_DOUBLE, ESP_MONEY (converted from SQL_NUMERIC)
SQL_BINARY	ESP_UTF8STR

Publisher SQL Data Type	Event Stream Processor Data Types
SQL_VARBINARY	ESP_UTF8STR
SQL_LONGVARBINARY	ESP_UTF8STR

Connectivity to Netezza Databases

Netezza drivers are not shipped with SAS Event Stream Processing. Nevertheless, you can use the database connector with a Netezza database table by following these steps:

1. Install the appropriate Netezza driver for your target platform.
2. Modify the **odbc.ini** file to include data source **"NZSQL = NetezzaSQL"**, and add a corresponding **[NZSQL]** section.
3. Modify the **odbcinist.ini** file to add **"NetezzaSQL = Installed"** to the **[ODBC Drivers]** section, and add a corresponding **[NetezzaSQL]** section.
4. Ensure that **odbc.ini** and **odbcinst.ini** are in the same directory.
5. Set the value of the **ODBCINI** environment variable to **"<*path*>odbc.ini"**.
6. Set the value of the **NZ_ODBC_INI_PATH** environment variable to **"<*path_to_odbc.ini*>"**.
7. Set the value of the database connector **connectstring** parameter to **"DSN=NZSQL;uid=<*uid*>;pwd=<*pwd*>"**.

Using File and Socket Connectors

Overview to File and Socket Connectors

File and socket connectors support both publish and subscribe operations on files or socket connections that stream the following data types:

- **binary**
- **csv**
- **xml**
- **json**
- **syslog** (only supports publish operations)
- **sashdat** (only supports subscribe operations, and only as a client type socket connector)

The file or socket nature of the connector is specified by the form of the configured **fsname**. A name in the form of *host*: *port* is a socket connector. Otherwise, it is a file connector.

When the connector implements a socket connection, it might act as a client or server, regardless of whether it is a publisher or subscriber. When you specify both *host* and *port* in the **fsname**, the connector implements the client. The configured host and port

specify the network peer implementing the server side of the connection. However, when *host* is blank (that is, when **fsname** is in the form of ": *port*"), the connection is reversed. The connector implements the server and the network peer is the client.

Use the following parameters when you specify file and socket connectors.

Table 16.5 *Required Parameters for File and Socket Connectors*

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

Table 16.6 *Required Parameters for Subscriber File and Socket Connectors*

Parameter	Description
snapshot	Specifies whether to send snapshot data.

Table 16.7 *Optional Parameters for Subscriber File and Socket Connectors*

Parameter	Description
collapse	Converts UPDATE_BLOCK events to UPDATE events in order to make subscriber output publishable. The default value is disabled.
periodicity	Specifies the interval in seconds at which the subscriber output file is closed and a new output file opened. When configured, a timestamp is appended to all output filenames for when the file was opened. This parameter does not apply to socket connectors with **fstype** other than **hdat**.
maxfilesize	Specifies the maximum size in bytes of the subscriber output file. When reached, a new output file is opened. When configured, a timestamp is appended to all output filenames. This parameter does not apply to socket connectors with **fstype** other than **hdat**.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.

Parameter	Description
hdatfilename	Specifies the name of the Objective Analysis Package Data (HDAT) file to be written to the Hadoop Distributed File System (HDFS). Include the full path, as shown in the HDFS browser when you browse the name node specified in the **fsname** parameter. Do not include the **.sashdat** extension. Applies only to and is required for the **hdat** connector.
hdfsblocksize	Specifies in Mbytes the block size used to write a Objective Analysis Package Data (HDAT) file. Applies only to and is required for the **hdat** connector.
hdatmaxstringlength	Specifies in bytes the fixed size of string fields in Objective Analysis Package Data (HDAT) files. You must specify a multiple of 8. Strings are padded with spaces. Applies only to and is required for the **hdat** connector.
hdatnumthreads	Specifies the size of the thread pool used for multi-threaded writes to data node socket connectors. A value of 0 or 1 indicates that writes are single-threaded and use only a name-node connection. Applies only to and is required for the **hdat** connector.
hdatmaxdatanodes	Specifies the maximum number of data node connectors. The default value is the total number of live data nodes known by the name mode. This parameter is ignored when **hdatnumthreads** <=1. Applies only to the **hdat** connector.
hdfsnumreplicas	Specifies the number of Hadoop Distributed File System (HDFS) replicas created with writing a Objective Analysis Package Data (HDAT) file. The default value is 1. Applies only to the **hdat** connector.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**
hdatlasrhostport	Specifies the SAS LASR Analytic Server host and port. Applies only to the **hdat** connector.
hdatlasrkey	Specifies the path to tklasrkey.sh. Applies only to the **hdat** connector.

Table 16.8 *Optional Parameters for Publisher File and Socket 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.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.
growinginputfile	Enables reading from a growing input file by publishers. The default value is disabled. When enabled, the publisher reads indefinitely from the input file until the connector is stopped or the server drops the connection. This parameter does not apply to socket connectors.
maxevents	Specifies the maximum number of events to publish.
prebuffer	Controls whether event blocks are buffered to an event block vector before doing any injects. The default value is false. Not valid with **growinginputfile** or for a socket connector
header	Specifies the number of input lines to skip before starting publish operations. The default value is 0. This parameter only applies to csv and syslog publish connectors.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**
ignorecsvparseerrors	Specifies that when a field in an input CSV file cannot be parsed, a null value is inserted, and publishing continues.

XML File and Socket Connector Data Format

The following XML elements are valid:

- **<data>**
- **<events>**
- **<transaction>**
- **<event>**
- **<opcode>**
- **<flags>**

In addition, any elements that correspond to event data field names are valid when contained within an event. Required elements are **<events>** and **<event>**, where **<events>** contains one or more **<event>** elements. This defines an event block. Events included within a **<transaction>** element are included in an event block with

type = transactional. Otherwise, events are included in event blocks with **type = normal**.

A single **<data>** element always wraps the complete set of event blocks in output XML. A closing **<data>** element on input XML denotes the end of streamed event blocks. Events are created from input XML with **opcode=UPSERT** and **flags=NORMAL**, unless otherwise denoted by an **<opcode>** or **<flags>** element.

Valid data for the **opcode** element in input data includes the following:

"opcode" Tag Value	Opcode
"**i** "	**Insert**
"**u**"	**Update**
"**p**"	**Upsert**
"**d**"	**Delete**
"**s**"	**Safe delete**

Valid data for the **flags** element in input data includes the following:

- "**n**" — the event is normal
- "**p**" — the event is partial update

The subscriber writes only Insert, Update, and Delete opcodes to the output data.

Non-key fields in the event are not required in input data, and their **value = NULL** if missing, or if the fields contain no data. The subscriber always writes all event data fields.

Any event field can include the "**partialupdate**" XML attribute. If its value is "**true**", and the event contains a **flags** element that specifies partial update, the event is built with the partial update flag. The field is flagged as a partial update with a null value.

JSON File and Socket Connector Data Format

A JSON publisher requires that the input JSON text conform to the following format:

`[[`*event_block_n*`],[`*event_block_n+1*`],[`*event_block_n+2*`],....]`

The outermost brackets define an array of event blocks. Each nested pair of brackets defines an array of events that corresponds to an event block. By default,the event block is **type= normal**. When the array is contained within an additional array (that is, an extra pair of brackets) the event block is **type= transactional**.

All field names are assumed to correspond to field names in the source window schema. There is one exception: when an "opcode" string field is present, it is used to set the event opcode. The value of the opcode string field must match one of the valid opcode values for XML or its uppercase equivalent. When not present, the event is built using opcode = Upsert. When the remaining fields do not match fields in the source window schema, the inject operation fails.

A JSON subscriber writes JSON text in the same format expected by the JSON publisher. In addition to the schema fields, each event contains an **opcode** field whose value is the event opcode. Each event block in the output JSON is separated by a comma and new line.

Syslog File and Socket Connector Notes

The syslog file and socket connector is supported only for publisher operations. It collects syslog events, converts them into ESP event blocks, and injects them into one or more source windows in a running ESP model.

The input syslog events are read from a file or named pipe written by the syslog daemon. Syslog events filtered out by the daemon are not seen by the connector. When reading from a named pipe, the following conditions must be met:

- The connector must be running in the same process space as the daemon.
- The pipe must exist.
- The daemon must be configured to write to the named pipe.

You can specify the **growinginputfile** parameter to read data from the file or named pipe as it is written.

The connector reads text data one line at a time, and parses the line into fields using spaces as delimiters.

The fields in the corresponding window schema must be of type ESP_UTF8STR or ESP_DATETIME. The number of schema fields must not exceed the number of fields parsed in the syslog file.

HDAT Subscribe Socket Connector Notes

This connector writes Objective Analysis Package Data (HDAT) files in SASHDAT format. This socket connector connects to the name node specified by the **fsname** parameter. The default LASR name node port is 15452. You can configure this port. Refer to the SAS Hadoop plug-in property for the appropriate port to which to connect.

The SAS Hadoop plug-in must be configured to permit puts from the service. Configure the property **com.sas.lasr.hadoop.service.allow.put=true**. By default, these puts are enabled. Ensure that this property is configured on data nodes in addition to the name node.

If run multi-threaded (as specified by the **hdatnumthreads** parameter), the connector opens socket connections to all the data nodes that are returned by the name node. It then writes subscriber event data as Objective Analysis Package Data (HDAT) file parts to all data nodes using the block size specified by the **hdfsblocksize** parameter. These file parts are then concatenated to a single file when the name node is instructed to do so by the connector.

The connector automatically concatenates file parts when stopped. It might also stop periodically as specified by the optional **periodicity** and **maxfilesize** parameters.

By default, socket connections to name nodes and to data nodes have a default time out of five minutes. This time out causes the server to disconnect the event stream processing connector after five minutes of inactivity on the socket. It is recommended that you disable the time out by configuring a value of 0 on the SAS Hadoop plug-in configuration property **com.sas.lasr.hadoop.socket.timeout**.

Because SASHDAT format consists of static row data, the connector ignores Delete events. It treats Update events the same as Insert events.

When you specify the **lasrhostport** parameter, the HDAT file is written and then the file is loaded into a LASR in-memory table. When you specify the **periodicity** or **maxfilesize** parameters, each write of the HDAT file is immediately followed by a load of that file.

Using the IBM WebSphere MQ Connector

The IBM WebSphere MQ connector (MQ) supports the IBM WebSphere Message Queue Interface for publish and subscribe operations. The subscriber receives event blocks and publishes them to an MQ queue. The publisher is an MQ subscriber, which injects received event blocks into source windows.

The IBM WebSphere MQ Client run-time libraries must be installed on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (for example, specifying **LD_LIBRARY_PATH** on Linux platforms).

The connector operates as an MQ client. It requires that you define the environment variable **MQSERVER** to specify the connector's MQ connection parameters. This variable specifies the server's channel, transport type, and host name. For more information, see your WebSphere documentation.

The topic string used by an MQ connector is a required connector parameter. In addition, an MQ subscriber requires a parameter that defines the message format used to publish events to MQ. The format options are csv or binary. An MQ publisher can consume any message type that is produced by an MQ subscriber.

An MQ publisher requires two additional parameters that are related to durable subscriptions. The publisher always subscribes to an MQ topic using a durable subscription. This means that the publisher can re-establish a former subscription and receive messages that had been published to the related topic while the publisher was disconnected.

These parameters are as follows:

- subscription name, which is user supplied and uniquely identifies the subscription
- subscription queue, which is the MQ queue 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 16.9 *Required Parameters for Subscriber MQ Connectors*

Parameter	Description
type	Specifies to subscribe.
mqtopic	Specifies the MQ topic name.
mqtype	Specifies binary, CSV, or JSON.
snapshot	Specifies whether to send snapshot data.

Table 16.10 *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.
`mqtype`	Specifies binary, CSV, or JSON.

Table 16.11 *Optional Parameters for Subscriber MQ Connectors*

Parameter	Description
`collapse`	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
`queuemanager`	Specifies the MQ queue manager.
`dateformat`	Specifies the date format. The default value is **`"%Y-%m-%d %H:%M:%S"`**.
`rmretdel`	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
`configfilesection`	Specifies the name of the section in **`/etc/connectors.config`** to parse for configuration parameters. Specify the value as **`[`*`configfilesection`*`]`**.
`protofile`	Specifies the **`.proto`** file that contains the Google Protocol Buffers message definition. This definition is used to convert event blocks to protobuf messages. When you specify this parameter, you must also specify the **`protomsg`** parameter.
`protomsg`	Specifies the name of a Google Protocol Buffers message in the **`.proto`** file that you specified with the **`protofile`** parameter. Event blocks are converted into this message.

Table 16.12 *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.

Parameter	Description
transactional	Sets the event block type to transactional. The default value is normal.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.
queuemanager	Specifies the MQ queue manager.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
ignorecsvparseerrors	When a field in an input CSV event cannot be parsed, insert a null value and continue publishing.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition. This definition is used to convert event blocks to protobuf messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.

Using the PI Connector

The PI Connector supports publish and subscribe operations for a PI Asset Framework (AF) server. The model must implement a window of a fixed schema to carry values that are associated with AF attributes owned by AF elements in the AF hierarchy. The AF data reference can be defined as a PI Point for these elements.

Note: Support for the PI Connector is available only on 64–bit Microsoft Windows platforms.

The PI Asset Framework (PI AF) Client from OSIsoft must be installed on the Microsoft Windows platform that hosts the running instance of the connector. The connector loads the OSIsoft.AFSDK.DLL public assembly, which requires .NET 4.0 installed on the target platform. The run-time environment must define the path to OSIsoft.AFSDK.dll.

The Microsoft Visual C++ Redistributable for Visual Studio 2012 package must be installed on the Microsoft Windows platform. The library for this connector is built with a different version of tools than those used by SAS Event Stream Processing libraries in order to achieve .NET 4.0 compatibility.

The window that is associated with the connector must use the following schema:

```
ID*:int32,elementindex:string,element:string,attribute:string,value:variable:,
timestamp:stamp,status:string
```

Use the following schema values.

Schema Value	Description
elementindex	Value of the connector *afelement* configuration parameter. Specify either an AF element name or an AF element template name.
element	Name of the AF element associated with the *value*.
attribute	Name of the AF attribute owned by the AF *element*.
value	Value of the AF *attribute*.
timestamp	Timestamp associated with the *value*.
status	Status associated with the *value*.

Note: The type of the *value* field is determined by the type of the AF attribute as defined in the AF hierarchy.

The following mapping of event stream processor data type to AF attributes is supported:

Event Stream Processor Data Type	AF Attribute
ESP_DOUBLE	TypeCode::Single TypeCode::Double
ESP_INT32	TypeCode::Byte TypeCode::SByte TypeCode::Char TypeCode::Int16 TypeCode::UInt16 TypeCode::Int32 TypeCode::UInt32
ESP_INT64	TypeCode::Int64 TypeCode::UInt64
ESP_UTF8STR	TypeCode::String
ESP_TIMESTAMP	TypeCode::DateTime

If an attribute has **`TypeCode::Object`**, the connected uses the type of the underlying PI point. Valid event stream processing data types to PI point mappings are as follows:

Event Stream Processor Data Type	PI Point
ESP_INT32	PIPointType::Int16 PIPointType::Int32
ESP_DOUBLE	PIPointType::Float16 PIPointType::Float32
ESP_TIMESTAMP	PIPointType::Timestamp
ESP_UTF8STR	PIPointType::String

Use the following parameters with PI connectors:

Table 16.13 *Required Parameters for Subscriber PI Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`afelement`	Specifies the AF element or element template name. Wildcards are supported.
`iselementtemplate`	Specifies whether the **`afelement`** parameter is an element template name. By default, the **`afelement`** parameter specifies an element name. Valid values are **`TRUE`** or **`FALSE`**.
`snapshot`	Specifies whether to send snapshot data.

Table 16.14 *Required Parameters for Publisher PI Connectors*

Parameter	Description
`type`	Specifies to publish.
`afelement`	Specifies the AF element or element template name. Wildcards are supported.
`iselementtemplate`	Specifies that the **`afelement`** parameter is an element template name. By default, the **`afelement`** parameter specifies an element name.

Table 16.15 *Optional Parameters for Subscriber PI Connectors*

Parameter	Description
rmretdel	Remove all delete events from event blocks received by the subscriber that were introduced by a window retention policy.
pisystem	Specifies the PI system. The default is the PI system that is configured in the PI AF client.
afdatabase	Specifies the AF database. The default is the AF database that is configured in the PI AF client.
afrootelement	Specifies the root element in the AF hierarchy from which to search for parameter **afelement**. The default is the top-level element in the AF database.
afattribute	Specifies a specific attribute in the element. The default is all attributes in the element.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.

Table 16.16 *Optional Parameters for Publisher PI Connectors*

Parameter	Description
blocksize	Specifies the number of events to include in a published event block. The default value is 1.
transactional	Sets the event block type to transactional. The default value is normal.
pisystem	Specifies the PI system. The default is the PI system that is configured in the PI AF client.
afdatabase	Specifies the AF database. The default is the AF database that is configured in the PI AF client.
afrootelement	Specifies the root element in the AF hierarchy from which to search for the parameter **afelement**. The default is the top-level element in the AF database.
afattribute	Specifies a specific attribute in the element. The default is all attributes in the element.
archivetimestamp	Specifies that all archived values from the specified timestamp onwards are to be published when connecting to the PI system. The default is to publish only new values.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.

Using the Project Publish Connector

Use the Project publish connector to subscribe to event blocks that are produced by a window from a different project within the event stream processing model. The connector receives that window's event blocks and injects them into its associated window. Window schemas must be identical. When the source project is stopped, the flow of event blocks to the publisher is stopped.

Table 16.17 *Required Parameters for the Project Publish Connector*

Parameter	Description
`type`	Specifies to publish.
`srcproject`	Specifies the name of the source project.
`srccontinuous query`	Specifies the name of the source continuous query.
`srcwindow`	Specifies the name of the source window.

Table 16.18 *Optional Parameters for the Project Publish Connector*

Parameter	Description
`maxevents`	Specifies the maximum number of events to publish.
`configfilesection`	Specifies the name of the section in **`/etc/connectors.config`** to parse for configuration parameters. Specify the value as **`[`*`configfilesection`*`]`**.

Using the Rabbit MQ Connector

The Rabbit MQ connector communicates with a Rabbit MQ server for publish and subscribe operations. The bus connectivity provided by the connector eliminates the need for the engine to manage individual publish/subscribe connections. The connector achieves a high capacity of concurrent publish/subscribe connections to a single engine.

A Rabbit MQ subscriber connector receives event blocks and publishes them to a Rabbit MQ routing key. A Rabbit MQ publisher connector reads event blocks from a dynamically created Rabbit MQ queue and injects them into an event stream processing source window.

Event blocks as transmitted through the Rabbit MQ server can be encoded as binary, CSV, Google protobufs, or JSON messages. The connector performs any conversion to and from binary format. The message format is a connector configuration parameter.

The Rabbit MQ connector supports hot failover operation. This mode requires that you install the presence-exchange plug-in on the Rabbit MQ server. You can download that plug-in from https://github.com/tonyg/presence-exchange.

A corresponding event stream processing publish/subscribe client plug-in library is available. This library enables a standard event stream processing publish/subscribe client application to exchange event blocks with an event stream processing server through a Rabbit MQ server. The exchange takes place through the Rabbit MQ server instead of through direct TCP connections. To enable this exchange, add a call to **`C_dfESPpubsubSetPubsubLib()`**.

When configured for hot failover operation, the active/standby status of the connector is coordinated with the Rabbit MQ server. Thus, a standby connector becomes active when the active connector fails. All involved connectors must meet the following conditions to guarantee successful switchovers:

- They must belong to identical ESP models.
- They 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. This is due to buffering of messages by standby connectors and coordination of the resumed flow with the Rabbit MQ server. The size of the message buffer is a required parameter for subscriber connectors.

You can configure a subscriber Rabbit MQ connector to send a custom snapshot of window contents to any subscriber client. The client must have established a new connection to the Rabbit MQ server. This enables late subscribers to catch up upon connecting. This functionality also requires that you install the presence-exchange plug-in on the Rabbit MQ server.

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 engine associated with that *host:port* combination. Metadata responses consist of some combination of the following:

- project name of the window associated with the connector
- query name of the window associated with the connector
- window name of the window associated with the connector
- the serialized schema of the window

You must install Rabbit MQ client run-time libraries on the platform that hosts the running instance of the connector. The connector uses the **`rabbitmq-c`** C libraries, which you can download from https://github.com/alanxz/rabbitmq-c. The run-time environment must define the path to those libraries (for example, specifying **`LD_LIBRARY_PATH`** on Linux platforms).

For queues that are created by a publisher, the optional **`buspersistence`** parameter controls both **`auto-delete`** and **`durable`**.

Setting of the `buspersistence` parameter	Effect
`true`	`auto-delete = false` `durable = true`

Setting of the `buspersistence` parameter	Effect
`false`	`auto-delete = true` `durable = false`

The following holds when consuming from those queues:

Setting of the `buspersistence` parameter	Effect
`true`	`exclusive = true` `noack = false`
`false`	`exclusive = false` `noack = true`

When the publisher connector creates a durable receive queue with auto-delete disabled, it consumes from that queue with **`noack = false`** but does not explicitly acknowledge messages. This enables a rebooted event stream processing server to receive persisted messages. The queue name is equal to the **`buspersistencequeue`** parameter appended with the configured topic parameter. The **`buspersistencequeue`** parameter must be unique on all publisher connectors that use the same Rabbit MQ exchange. The publisher connector enforces this by consuming the queue in exclusive mode when **`buspersistence`** is enabled.

For a subscriber connector, enabling **`buspersistence`** means that messages are sent with the delivery mode set to **`persistent`**.

For exchanges that are created by a publish or a subscribe, **`buspersistence`** controls only **`durable`**. That is, when **`buspersistence = true`**, **`durable = true`**, and when **`buspersistence = false`**, **`durable = false`**.

Table 16.19 *Required Parameters for Subscriber Rabbit MQ Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`rmquserid`	Specifies the user name required to authenticate the connector's session with the Rabbit MQ server.
`rmqpassword`	Specifies the password associated with **`rmquserid`**.
`rmqhost`	Specifies the Rabbit MQ server host name.
`rmqport`	Specifies the Rabbit MQ server port.
`rmqexchange`	Specifies the Rabbit MQ exchange created by the connector, if nonexistent.

Parameter	Description
rmqtopic	Specifies the Rabbit MQ routing key to which messages are published.
rmqtype	Specifies **binary**, **CSV**, or **JSON**.
urlhostport	Specifies the *host:port* field in the metadata topic subscribed to on start-up to field metadata requests.
numbufferedmsgs	Specifies the maximum number of messages buffered by a standby subscriber connector. When exceeded, the oldest message is discarded. When the connector goes active, the buffer is flushed and buffered messages are sent to the fabric as required to maintain message ID sequence.
snapshot	Specifies whether to send snapshot data.

Table 16.20 *Required Parameters for Publisher Rabbit MQ Connectors*

Parameter	Description
type	Specifies to publish.
rmquserid	Specifies the user name required to authenticate the connector's session with the Rabbit MQ server.
rmqpassword	Specifies the password associated with **rmquserid**.
rmqhost	Specifies the Rabbit MQ server host name.
rmqport	Specifies the Rabbit MQ server port.
rmqexchange	Specifies the Rabbit MQ exchange created by the connector, if nonexistent.
rmqtopic	Specifies the Rabbit MQ routing key to which messages are published.
rmqtype	Specifies **binary**, **CSV**, or **JSON**.
urlhostport	Specifies the *host:port* field in the metadata topic subscribed to on start-up to field metadata requests.

Table 16.21 *Optional Parameters for Subscriber Rabbit MQ Connectors*

Parameter	Description
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is **disabled**.

Parameter	Description
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
hotfailover	Enables hot failover mode.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.
buspersistence	Specify to send messages using persistent delivery mode.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition used to convert event blocks to **protobuf** messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.
csvincludeschema	Specifies **"never"**, **"once"**, or **"pereventblock"**. The default value is **"never"**. When **rmqtype = CSV**, prepend output CSV with the window's serialized schema.
useclientmsgid	When performing a failover operation and extracting a message ID from an event block, use the client-generated message ID instead of the engine-generated message ID.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]** .

Table 16.22 *Optional Parameters for Publisher Rabbit MQ Connectors*

Parameter	Description
transactional	When **rmqtype = CSV**, sets the event block type to **transactional**. The default value is **normal**.
blocksize	When **rmqytype = CSV**, specifies the number of events to include in a published event block. The default value is 1.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.
buspersistence	Controls both **auto-delete** and **durable**.
buspersistencequeue	Specify the queue name used by a persistent publisher.
ignorecsvparseerrors	Specifies that when a field in an input CSV file cannot be parsed, a null value is inserted, and publishing continues.

Parameter	Description
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition used to convert event blocks to **protobuf** messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.

Using the SMTP Subscribe Connector

You can use the Simple Mail Transfer Protocol (SMTP) subscribe connector to e-mail window event blocks or single events, such as alerts or items of interest. This connector is subscribe-only. The connection to the SMTP server uses port 25. No user authentication is performed, and the protocol runs unencrypted.

The e-mail sender and receiver addresses are required information for the connector. The e-mail subject line contains a standard event stream processor URL in the form "**dfESP://*host*:*port*/*project*/*contquery*/*window***", followed by a list of the key fields in the event. The e-mail body contains data for one or more events encoded in CSV format.

The parameters for the SMTP connector are as follows:

Table 16.23 *Required Parameters for the SMTP Connector*

Parameter	Description
type	Specifies to 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.
snapshot	Specifies whether to send snapshot data.

Table 16.24 Optional Parameters for SMTP Connectors

Parameter	Description
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
emailperevent	Specifies true or false. The default is false. If false, each e-mail body contains a full event block. If true, each mail body contains a single event.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.

Using the Sniffer Connector

The sniffer connector captures packets from a local network interface in promiscuous mode and builds an event per received packet to be injected into a source window. An instance of the connector is configured with the interface name, a protocol, and a comma separated list of fields to be extracted and included in the event. Additional connectors can be instanced to capture additional packets in any combination of interface/protocol/fields, and injected to any source window.

Protocol support is currently limited to the following:

- HTTP packets sent to port 80 over TCP
- Radius Accounting-Request packets sent to port 1813 over UDP. Only attribute values between 1 and 190 inclusive are supported. If you capture the **Attribute-Specific** field of a **Vendor-Specific** attribute, you must configure the **vendorid** and **vendortype** connector parameters.

Support is included for an optional 802.1Q VLAN tag header following the Ethernet header, but in all other cases the IP header must directly follow the Ethernet header.

The connector uses the **libpcap** libraries, which are not shipped with ESP. You must install these libraries separately on the target machine. They are available for download at http://www.tcpdump.orgor, for Microsoft Windows, http://www.winpcap.org.

Most kernels protect against applications opening raw sockets. On Linux platforms, the connector logs the following error message unless the application is given permission to open raw sockets:

```
"You don't have permission to capture on that device (socket: Operation not permitted)"
```

To grant suitable permissions to the server that is running the connector, run the following command: **setcap cap_net_raw,cap_net_admin=eip *executable***. You must have root privileges to run the command.

A side effect of granting these permissions is that the application no longer uses the shell's **LD_LIBRARY_PATH** environment variable. For the SAS Event Stream

Processing server application, this means that it must have an alternative method of finding shared objects in **$DFESP_HOME/lib**. Use the **ldconfig** command to update the shared library cache with the **$DFESP_HOME/lib** directory before running the SAS Event Stream Processing server.

The **packetfields** configuration parameter uses SAS Event Stream Processing schema-like syntax, and must match the source window schema. There are two exceptions:

- The source window schema must contain an additional **index:int64** field that contains an increasing index value and that serves as the key field.
- The source window schema must also contain an additional**frame_time:stamp** field that contains the timestamp created by the Pcap driver.
- When the optional **addtimestamp** connector configuration parameter is specified, the source window schema must also contain a **ts:stamp** field to hold the timestamp. This field is created by the connector and holds the current time.

The **index** and **frame_time** fields must be the first two fields in the window schema. The **ts** field must be the last field in the window schema.

When the **blocksize** parameter is configured with a value greater than 1, the connector builds event blocks of size no greater than the configured **blocksize**. It can build event blocks with a smaller size when the **pcap** driver buffer has filled up, or when the read **timeout** on the socket opened by the **pcap** driver has expired. This ensures that the connector injects all events available from packets received on the interface as soon as possible, without having to wait to fill an event block.

When a received packet is malformed or contains an invalid parameter or length, the connector generally logs an **info** level message, ignores the packet, and continues.

For testing, you can receive packets from a **.pcap** capture file instead of a network interface. You can specify the name of this capture file in the connector interface parameter. When the connector cannot find a matching interface name, it treats the name as a **.pcap** filename.

Table 16.25 *Required Parameters*

Parameter	Definition
type	Specifies to publish. The required value is **pub**.
interface	Specifies the name of the network interface on the local machine from which to capture packets.
protocol	Specifies the protocol type of packets to be captured. Currently supported values are **radius** and **http**.

Parameter	Definition
packetfields	Specifies the packet fields to be extracted from a captured packet and included in the published event. Use SAS Event Stream Processing schema syntax. This value does not include the **index:int64 frame_time:stamp**, or **ts:stamp** fields. Separate nested fields with an underscore character. Match field names to standard names as displayed by the Wireshark open-source packet analyzer. You must remove spaces and hyphens to maintain field name integrity. An example for Radius is as follows: **"radius_AcctStatusType:int32,radius_EventTimestamp:stamp,radius_FramedIPAddress:string,radius_UserName:string"**. An example for HTTP is as follows: **"ip_Source:string,ip_Destination:string,http_Host:string,http_Referer:string,http_UserAgent:string,http_GET_RequestURI:string"**.

Table 16.26 *Optional Parameters*

Parameter	Description
transactional	Sets the event block type to transactional. The default value is normal.
blocksize	Specifies the number of events to include in a published event block. The default value is 1.
addtimestamp	Specifies to append an **ESP_TIMESTAMP** field to each published event. The field value is the current time when the packet was received by the connector. This field must be present in the source window schema, but it is not required in the connector **packetfields** parameter.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
vendorid	Specifies the **vendor-Id** field to match when capturing the **Attribute-Specific** field in a **Vendor-Specific** attribute in a Radius Accounting-Request packet.

Parameter	Description
vendortype	Specifies the **vendor-Type** field to match when capturing the **Attribute-Specific** field in a **Vendor-Specific** attribute in a Radius Accounting-Request packet.

Using the Solace Systems Connector

The Solace Systems connector communicates with a hardware-based Solace fabric for publish and subscribe operations.

A Solace subscriber connector receives event blocks and publishes them to this Solace topic:

"*host*:*port*/*projectname*/*queryname*/*windowname*/O

A Solace publisher connector reads event blocks from the following Solace topic

"*host*:*port*/*projectname*/*queryname*/*windowname*/I

and injects them into the corresponding source window.

As a result of the bus connectivity provided by the connector, the 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.

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 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 *buspersistenceque* 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.
- Google protocol buffer support must not be enabled, because these binary messages do not contain a usable message ID.

When a new subscriber connector becomes active, outbound message flow remains synchronized due to buffering of messages by standby connectors and coordination of the resumed flow with the fabric. The size of this message buffer is a required parameter for subscriber connectors.

Solace connectors can be configured to use a persistent mode of messaging instead of the default direct messaging mode. (See the description of the **`buspersistence`** configuration parameter.) This mode might require regular purging of persisted data by an administrator, if there are no other automated mechanism to age out persisted messages. The persistent mode reduces the maximum throughput of the fabric, but it enables a publisher connector to connect to the fabric after other connectors have already processed data. The fabric updates the connector with persisted messages and synchronizes window states with the other engines in a hot failover group.

Solace subscriber connectors subscribe to a special topic that enables them to be notified when a Solace client subscribes to the connector's topic. When the connector is configured with snapshot enabled, it sends a custom snapshot of the window contents to that client. This enables late subscribers to catch up upon connecting.

Solace connector configuration parameters named "sol…" are passed unmodified to the Solace API by the connector. See your Solace documentation for more information about these parameters.

Use the following parameters with Solace Systems connectors

Table 16.27 *Required Parameters for Subscriber Solace Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`soluserid`	Specifies the user name required to authenticate the connector's session with the appliance.
`solpassword`	Specifies the password associated with **`soluserid`**.

Parameter	Description
solhostport	Specifies the appliance to connect to, in the form "*host*:*port*".
solvpn	Specifies the appliance message VPN to assign the client to which the session connects.
soltopic	Specifies the Solace destination topic to which to publish.
urlhostport	Specifies the *host*:*port* field in the metadata topic subscribed to on start-up to field metadata requests.
numbufferedmsgs	Specifies the maximum number of messages buffered by a standby subscriber connector. If exceeded, the oldest message is discarded. If the connector goes active the buffer is flushed, and buffered messages are sent to the fabric as required to maintain message ID sequence.
snapshot	Specifies whether to send snapshot data.

Table 16.28 *Required Parameters for Publisher Solace Connectors*

Parameter	Description
type	Specifies to publish.
soluserid	Specifies the user name required to authenticate the connector's session with the appliance.
solpassword	Specifies the password associated with **soluserid**.
solhostport	Specifies the appliance to connect to, in the form "*host*:*port*".
solvpn	Specifies the appliance message VPN to assign the client to which the session connects.
soltopic	Specifies the Solace topic to which to subscribe.
urlhostport	Specifies the *host*:*port* field in the metadata topic subscribed to on start-up to field metadata requests.

Table 16.29 *Optional Parameters for Subscriber Solace Connectors*

Parameter	Description
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
hotfailover	Enables hot failover mode.

Parameter	Description
buspersistence	Sets the Solace message delivery mode to Guaranteed Messaging. The default value is Direct Messaging.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition used to convert event blocks to **protobuf** messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
json	Enables transport of event blocks encoded as JSON messages.

Table 16.30 *Optional Parameters for Publisher Solace Connectors*

Parameter	Description
buspersistence	Creates the Guaranteed message flow to bind to the topic endpoint provisioned on the appliance that the published Guaranteed messages are delivered and spooled to. By default this flow is disabled, because it is not required to receive messages published using Direct Messaging.
buspersistencequeue	Specifies the name of the queue to which the Guaranteed message flow binds.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition used to convert event blocks to **protobuf** messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
json	Enables transport of event blocks encoded as JSON messages.

Using the Teradata Connector

The Teradata connector uses the Teradata Parallel Transporter (TPT) API to support subscribe operations against a Teradata server.

The Teradata Tools and Utilities (TTU) package must be installed on the platform running the connector. The run-time environment must define the path to the Teradata client libraries in the TTU. For Teradata ODBC support while using the load operator, you must also define the path to the Teradata TeraGSS libraries in the TTU. To support logging of Teradata messages, you must define the **`NLSPATH`** environment variable appropriately. For example, with a TTU installation in **`/opt/teradata`**, define **`NLSPATH`** as "**`/opt/teradata/client/`*`version`*`/tbuild/msg64/%N`**". The connector has been certified using TTU version 14.10.

For debugging, you can install optional PC-based Teradata client tools to access the target database table on the Teradata server. These tools include the GUI SQL workbench (Teradata SQL Assistant) and the DBA/Admin tool (Teradata Administrator), which both use ODBC.

You must use one of three TPT operators to write data to a table on the server:**`stream`**, **`update`**, or **`load`**.

Operator	Description
`stream`	Works like a standard ESP database subscriber connector, but with improved throughput gained through using the TPT. Supports insert, update, and delete events. As it receives events from the subscribed window, it writes them to the target table. If you set the required **`tdatainsertonly`** configuration parameter to **`false`**, serialization is automatically enabled in the TPT to maintain correct ordering of row data over multiple sessions.
`update`	Supports insert/update/delete events but writes them to the target table in batch mode. The batch period is a required connector configuration parameter. At the cost of higher latency, this operator provides better throughput with longer batch periods (for example minutes instead of seconds).
`load`	Supports insert events. Requires an empty target table. Provides the most optimized throughput. Staggers data through a pair of intermediate staging tables. These table names and connectivity parameters are additional required connector configuration parameters. In addition, the write from a staging table to the ultimate target table uses the generic ODBC driver used by the database connector. Thus, the associated connect string configuration and odbc.ini file specification is required. The staging tables are automatically created by the connector. If the staging tables and related error and log tables already exist when the connector starts, it automatically drops them at start-up.

Table 16.31 *Required Parameters for Teradata Connectors*

Parameter	Description
type	Specifies to subscribe
desttablename	Target table name.
tdatausername	User name for the user account on the target Teradata server.
tdatauserpwd	User password for the user account on the target Teradata server.
tdatatdpid	Target Teradata server name
tdatamaxsessions	Maximum number of sessions created by the TPT to the Teradata server.
tdataminsessions	Minimum number of sessions created by the TPT to the Teradata server.
tdatadriver	The operator: **stream**, **update**, or **load**.
tdatainsertonly	Specifies whether events in the subscriber event stream processing window are insert only. Must be **true** when using the load operator.
snapshot	Specifies whether to send snapshot data.

Table 16.32 *Optional Parameters for Teradata Connectors*

Parameter	Description
rmretdel	Removes all delete events from event blocks received by the subscriber that were introduced by a window retention policy.
tdatabatchperiod	Specifies the batch period in seconds. Required when using the **update** operator, otherwise ignored.
stage1tablename	Specifies the first staging table. Required when using the **load** operator, otherwise ignored.
stage2tablename	Specifies the second staging table. Required when using the **load** operator, otherwise ignored.
connectstring	Specifies the connect string used to access the target and staging tables. Use the form **"DSN=*dsnname*;UID=*userid*;pwd=*password*"**. Required when using the **load** operator, otherwise ignored.

Parameter	Description
tdatatracelevel	Specifies the trace level for Teradata messages written to the trace file in the current working directory. The trace file is named *operator*1.txt. Default is 1 (TD_OFF). Other valid values are: 2 (TD_OPER), 3 (TD_OPER_CLI), 4 (TD_OPER_OPCOMMON), 5 (TD_OPER_SPECIAL), 6 (TD_OPER_ALL), 7 (TD_GENERAL), and 8 (TD_ROW).
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.

Using the Tervela Data Fabric Connector

The Tervela Data Fabric connector communicates with a software or hardware-based Tervela Data Fabric for publish and subscribe operations.

A Tervela subscriber connector receives events blocks and publishes to the following Tervela topic:
"SAS.ENGINES.*enginename*.*projectname*.*queryname*.*windowname*.OUT."

A Tervela publisher connector reads event blocks from the following Tervela topic: "SAS.ENGINES.*enginename*.*projectname*.*queryname*.*windowname*.IN" and injects them into source windows.

As a result of the bus connectivity provided by the Tervela Data Fabric connector, the 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.

You must install the Tervela run-time libraries on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (specify **LD_LIBRARY_PATH** on Linux platforms, for example).

The Tervela Data Fabric Connector has the following characteristics:

- It works with binary event blocks. No other event block formats are supported.
- It operates as a Tervela client. All Tervela Data Fabric connectivity parameters are required as connector configuration parameters.

Before using Tervela Data Fabric connectors, you must configure the following items on the Tervela TPM Provisioning and Management System:

- a client user name and password to match the connector's **tvauserid** and **tvapassword** configuration parameters
- the inbound and outbound topic strings and associated schema
- publish or subscribe entitlement rights associated with a client user name

When the connector starts, it publishes a message to topic SAS.META.*tvaclientname* (where *tvaclientname* is a connector configuration parameter). This message contains the following information:

- The mapping of the ESP engine name to a *host*:*port* field potentially used by an ESP publish/subscribe client. The *host*:*port* string is the required **urlhostport**

connector configuration parameter, and is substituted by the engine name in topic strings used on the fabric.

- The project, query, and window names of the window associated with the connector, as well as the serialized schema of the window.

All messaging performed by the Tervela connector uses the Tervela Guaranteed Delivery mode. Messages are persisted to a Tervela TPE appliance. When a publisher connector connects to the fabric, it receives messages already published to the subscribed topic over a recent time period. By default, the publisher connector sets this time period to eight hours. This enables a publisher to catch up with a day's worth of messages. Using this mode requires regular purging of persisted data by an administrator when there are no other automated mechanism to age out persisted messages.

Tervela subscriber connectors support a hot failover mode. The active/standby status of the connector is coordinated with the fabric so that a standby connector becomes active when the active connector fails. Several conditions must be met to guarantee successful switchovers:

- The engine names of the ESP engines running the involved connectors must all be identical. This set of ESP engines is called the failover group.
- All involved connectors must be active on the same set of topics.
- All involved subscriber connectors must be configured with the same **`tvaclientname`**.
- All involved connectors must initiate message flow at the same time, and with the TPE purged of all messages on related topics. This is required because message IDs must be synchronized across all connectors.
- Message IDs that are set by the injector of event blocks into the model must be sequential and synchronized with IDs used by other standby connectors. When the injector is a Tervela publisher connector, that connector sets the message ID on all injected event blocks, beginning with ID = 1.

When a new subscriber connector becomes active, outbound message flow remains synchronized due to buffering of messages by standby connectors and coordination of the resumed flow with the fabric. The size of this message buffer is a required parameter for subscriber connectors.

Tervela connector configuration parameters named **`tva...`** are passed unmodified to the Tervela API by the connector. See your Tervela documentation for more information about these parameters.

Use the following parameters with Tervela connectors:

Table 16.33 *Required Parameters for Subscriber Tervela Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`tvauserid`	Specifies a user name defined in the Tervela TPM. Publish-topic entitlement rights must be associated with this user name.
`tvapassword`	Specifies the password associated with **`tvauserid`**.
`tvaprimarytmx`	Specifies the host name or IP address of the primary TMX.

Parameter	Description
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.
tvamaxoutstand	Specifies the maximum number of unacknowledged messages that can be published to the Tervela fabric (effectively the size of the publication cache). Should be twice the expected transmit rate.
numbufferedmsgs	Specifies the maximum number of messages buffered by a standby subscriber connector. When exceeded, the oldest message is discarded. If the connector goes active the buffer is flushed, and buffered messages are sent to the fabric as required to maintain message ID sequence.
urlhostport	Specifies the "*host*/*port*" string sent in the metadata message published by the connector on topic SAS.META.*tvaclientname* when it starts.
snapshot	Specifies whether to send snapshot data.

Table 16.34 *Required Parameters for Publisher Tervela Connectors*

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

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 16.35 *Optional Parameters for Subscriber Tervela Connectors*

Parameter	Description
collapse	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.
hotfailover	Enables hot failover mode
tvasecondarytmx	Specifies the host name or IP address of the secondary TMX. Required if logging in to a fault-tolerant pair.
tvalogfile	Causes the connector to log to the specified file instead of to syslog (on Linux or Solaris) or Tervela.log (on Windows)
tvapubbwlimit	Specifies the maximum bandwidth, in Mbps, of data published to the fabric. The default value is 100 Mbps.
tvapubrate	Specifies the rate at which data messages are published to the fabric, in Kbps. The default value is 30,000 messages per second.
tvapubmsgexp	Specifies the maximum amount of time, in seconds, that published messages are kept in the cache in the Tervela API. This cache is used as part of the channel egress reliability window (if retransmission is required). The default value is 1 second.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.

Parameter	Description
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition. This definition is used to convert event blocks to protobuf messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.
json	Enables transport of event blocks encoded as JSON messages.

Table 16.36 *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)
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition. This definition is used to convert event blocks to protobuf messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.
json	Enables transport of event blocks encoded as JSON messages.

Using the Tibco Rendezvous (RV) Connector

The Tibco Rendezvous (RV) connector supports the Tibco RV API for publish and subscribe operations through a Tibco RV daemon. The subscriber receives event blocks and publishes them to a Tibco RV subject. The publisher is a Tibco RV subscriber, which injects received event blocks into source windows.

The Tibco RV run-time libraries must be installed on the platform that hosts the running instance of the connector. The run-time environment must define the path to those libraries (for example, specifying **`LD_LIBRARY_PATH`** on Linux platforms).

The system path must point to the **`Tibco/RV/bin`** directory so that the connector can run the RVD daemon.

The subject name used by a Tibco RV connector is a required connector parameter. A Tibco RV subscriber also requires a parameter that defines the message format used to publish events to Tibco RV. The format options are CSV or binary. A Tibco RV publisher can consume any message type produced by a Tibco RV subscriber.

By default, the Tibco RV connector assumes that a Tibco RV daemon is running on the same platform as the connector. Alternatively, you can specify the connector **`tibrvdaemon`** configuration parameter to use a remote daemon.

Similarly, you can specify the optional **`tibrvservice`** and **`tibrvnetwork`** parameters to control the Rendezvous service and network interface used by the connector. For more information, see your Tibco RV documentation.

The Tibco RV connector relies on the default multicast protocols for message delivery. The reliability interval for messages sent to and from the Tibco RV daemon is inherited from the value in use by the daemon.

Use the following parameters with Tibco RV connectors:

Table 16.37 *Required Parameters for Subscriber Tibco RV Connectors*

Parameter	Description
`type`	Specifies to subscribe.
`tibrvsubject`	Specifies the Tibco RV subject name.
`tibrvtype`	Specifies binary, CSV, or JSON.
`snapshot`	Specifies whether to send snapshot data.

Table 16.38 *Required Parameters for Publisher Tibco RV Connectors*

Parameter	Description
`type`	Specifies to publish.
`tibrvsubject`	Specifies the Tibco RV subject name.
`tibrvtype`	Specifies binary, CSV, or JSON.

Table 16.39 *Optional Parameters for Subscriber Tibco RV Connectors*

Parameter	Description
`collapse`	Enables conversion of UPDATE_BLOCK events to make subscriber output publishable. The default value is disabled.

Parameter	Description
tibrvservice	Specifies the Rendezvous service used by the Tibco RV transport created by the connector. The default service name is “rendezvous”.
tibrvnetwork	Specifies the network interface used by the Tibco RV transport created by the connector. The default network depends on the type of daemon used by the connector.
tibrvdaemon	Specifies the Rendezvous daemon used by the connector. The default is the default socket created by the local daemon.
rmretdel	Specifies to remove all delete events from event blocks received by a subscriber that were introduced by a window retention policy.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition. This definition is used to convert event blocks to protobuf messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.

Table 16.40 *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.
dateformat	Specifies the date format. The default value is **"%Y-%m-%d %H:%M:%S"**.
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.

Parameter	Description
tibrvdaemon	Specifies the Rendezvous daemon used by the connector. The default is the default socket created by the local daemon.
configfilesection	Specifies the name of the section in **/etc/connectors.config** to parse for configuration parameters. Specify the value as **[*configfilesection*]**.
ignorecsvparseerrors	When a field in an input CSV event cannot be parse, insert a null value and continue publishing.
protofile	Specifies the **.proto** file that contains the Google Protocol Buffers message definition. This definition is used to convert event blocks to protobuf messages. When you specify this parameter, you must also specify the **protomsg** parameter.
protomsg	Specifies the name of a Google Protocol Buffers message in the **.proto** file that you specified with the **protofile** parameter. Event blocks are converted into this message.

Writing and Integrating a Custom Connector

Writing a Custom Connector

When you write your own connector, the connector class must inherit from base class **dfESPconnector**.

Connector configuration is maintained in a set of key or value pairs where all keys and values are text strings. A connector can obtain the value of a configuration item at any time by calling **getParameter()** and passing the key string. An invalid request returns an empty string.

A connector can implement a subscriber that receives events generated by a window, or a publisher that injects events into a window. However, a single instance of a connector cannot publish and subscribe simultaneously.

A subscriber connector receives events by using a callback method defined in the connector class that is invoked in a thread owned by the engine. A publisher connector typically creates a dedicated thread to read events from the source. It then injects those events into a source window, leaving the main connector thread for subsequent calls made into the connector.

A connector must define these static data structures:

Static Data Structure	Description
dfESPconnectorInfo	Specifies the connector name, publish/subscribe type, initialization function pointer, and configuration data pointers.

Static Data Structure	Description
subRequiredConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing required configuration parameters for a subscriber.
sizeofSubRequiredConfig	Specifies the number of entries in **subRequiredConfig**.
pubRequiredConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing required configuration parameters for a publisher.
sizeofPubRequiredConfig	Specifies the number of entries in **pubRequiredConfig**.
subOptionalConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing optional configuration parameters for a subscriber.
sizeofSubOptionalConfig	Specifies the number of entries in **subOptionalConfig**.
pubOptionalConfig	Specifies an array of **dfESPconnectorParmInfo_t** entries listing optional configuration parameters for a publisher.
sizeofPubOptionalConfig	Specifies the number of entries in **pubOptionalConfig**.

A connector must define these static methods:

Static Method	Description
dfESPconnector *initialize(dfESPengine *_engine_, dfESPpsLib_t _psLib_)	Returns an instance of the connector.
dfESPconnectorInfo *getConnectorInfo()	Returns the **dfESPconnectorInfo** structure.

You can invoke these static methods before you create an instance of the connector.

A connector must define these virtual methods:

Virtual Method	Description
start()	Starts the connector. Must call base class method **checkConfig()** to validate connector configuration before starting. Must also call base class method **start()**. Must set **variable _started = true** upon success.
stop()	Stops the connector. Must call base class method **stop()**. Must leave the connector in a state whereby **start()** can be subsequently called to restart the connector.
callbackFunction()	Specifies the method invoked by the engine to pass event blocks generated by the window to which it is connected.
errorCallbackFunction()	Specifies the method invoked by the engine to report errors detected by the engine. Must call user callback function **errorCallback**, if nonzero.

A connector must set its running state for use by the connector orchestrator. It does this by calling the **dfESPconnector::setState()** method. The two relevant states are **state_RUNNING** and **state_FINISHED**. All connectors must set **state_RUNNING** when they start. Only connectors that actually finish transferring data need to set **state_FINISHED**. Typically, setting state in this way is relevant only for publisher connectors that publish a finite number of event blocks.

Finally, a derived connector can implement up to ten user-defined methods that can be called from an application. Because connectors are plug-ins loaded at run time, a user application cannot directly invoke class methods. It is not linked against the connector.

The base connector class defines virtual methods **userFunction_01** through **userFunction_10**, and a derived connector then implements those methods as needed. For example:

```
void * myConnector::userFunction_01(void *myData) {
```

An application would invoke the method as follows:

```
myRC = myConnector->userFunction_01((void *)myData);
```

Integrating a Custom Connector

All connectors are managed by a global connector manager. The default connectors shipped with SAS Event Stream Processing are automatically loaded by the connector manager during product initialization. Custom connectors built as libraries and placed in **$DFESP_HOME/lib/plugins** are also loaded during initialization, with the exception of those listed in **$DFESP_HOME/etc/connectors.excluded**.

After initialization, the connector is available for use by any event stream processor window defined in an application. As with any connector, an instance of it can be obtained by calling the window **getConnector()** method and passing its user-defined

method. You can configure the connector using **`setParameter()`** before starting the project.

Chapter 17

Using Adapters

Overview to Adapters 297
Using the Database Adapter 298
Using the Event Stream Processor Adapter 300
Using the File and Socket Adapter 300
Using the IBM WebSphere MQ Adapter 303
Using the HDAT Reader Adapter 304
Using the HDFS (Hadoop Distributed File System) Adapter 305
Using the Java Message Service (JMS) Adapter 308
Using the SAS LASR Analytic Server Adapter 311
Using the PI Adapter 314
Using the Rabbit MQ Adapter 315
Using the REST Subscriber Adapter 317
Using the SAS Data Set Adapter 318
Using the SMTP Subscriber Adapter 321
Using the Sniffer Publisher Adapter 322
Using the Solace Systems Adapter 323
Using the Teradata Subscriber Adapter 325
Using the Tervela Data Fabric Adapter 327
Using the Tibco Rendezvous (RV) Adapter 329
Using the Twitter Publisher Adapter 331

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 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.

Similar to connectors, adapters can obtain their configuration parameters from a file. Specify the section label in the file on the command line using the **-C** switch. For more information, see "Setting Configuration Parameters in a File" on page 252.

Note: Set file-based configuration for Java adapters in the file **/etc/javaadapters.config**.

You can find adapters in the following directory:

```
$DFESP_HOME/bin
```

Using the Database Adapter

The Database Adapter supports publish and subscribe operations on databases using DataDirect drivers. The adapter is certified for the following platforms:

- Oracle
- MySQL
- IBM DB2
- Greenplum
- PostgreSQL
- SAP Sybase ASE
- Teradata
- Microsoft SQL Server
- IBM Informix
- Sybase IQ

However, drivers exist for many other databases and appliances. This adapter requires that database connectivity be available by using a Data Source Name (DSN) configured in the odbc.ini file. The file is pointed to by the ODBCINI environment variable.

Note: The database adapter uses generic DataDirect ODBC drivers for the Teradata platform. A separately packaged adapter uses the Teradata Parallel Transporter for improved performance.

Subscriber usage:

dfesp_db_adapter -k sub -h *url* -c *connectstring*-t *tablename*
<-g *gdconfig*> <-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-C [*configfilesection*]><-x *commitrows*><-z *commitsecs*><-q>

Publisher usage:

dfesp_db_adapter -k pub —h *url* -c *connectstring* -s *selectstatement*
<-b *blocksize*> <-e> <-g *gdconfig*> <-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*> <-C [*configfilesection*]>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
–h ***url***	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***. Append **?snapshot=true** | **false** if needed. Append **?rmretdel=true** | **false** for subscribers if needed.
-c ***connectstring***	database connection string, in the form **"DSN=*data_source_name*<;uid=*userid*><;pwd=*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** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** 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.
–x ***commitrows***	Specifies the minimum number of rows to buffer.
–z ***commitsecs***	Specifies the maximum number of seconds to hold onto an incomplete commit buffer.
-q	Specifies to ignore errors when executing SQL Inserts, Updates, or Deletes.
-C [*configfilesection***]**	Specify the name of the section in **/etc/connectors.config** to parse for configuration parameters.

Using the Event Stream Processor Adapter

The event stream processor adapter enables you to subscribe to a window and publish what it passes into another source window. This operation can occur within a single event stream processor. More likely it occurs across two event stream processors that are run on different machines on the network.

No corresponding event stream processor connector is provided.

Usage:

dfesp_esp_adapter -s *url* –p *url*

Parameter	Definition
-s *url*	Specify the subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*contquery*/*window*?snapshot=true |false>?collapse=true | false**"
-p *url*	Specify the publish standard URL in the form **dfESP://*host*:*port*/*project*/*contquery*/*window***

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

Using the File and Socket Adapter

The File and Socket adapter supports publish and subscribe operations on files or socket connections that stream the following data types:

- dfESP binary
- CSV
- XML
- JSON
- syslog
- hdat

Subscriber use:

dfesp_fs_adapter -k sub - h *url* -f *fsname*-t binary | csv | xml | json | hdat
<-c *period*> <-s *maxfilesize*> <-d *dateformat*>
<-r *rate*> <-a *aggrsize*> <-n>
<-g *gdconfig*> <-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off> <-y *log_configfile*> <-o *hdat_filename*>
<-q *hdat_max_data_nodes*> <-u *hdfs_blocksize*> <-v *hdfs_numreplicas*>
<-w *hdat_numthreads*> <-z *hdat_max_stringlength*><-C [*configfilesection*]>
<-H *hdatlasrhostport*> <-K *hdatalasrkey*>

Publisher use:

dfesp_fs_adapter-k pub -h *url* -f *fsname*
-t binary | csv | xml | json | syslog <-b *blocksize*> <-d *dateformat*>
<-r *rate*> <-m *maxevents*> <-e> <-i> <-p> <-n> <-g *gdconfig*>
<-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *log_configfile*> <-x *header*> <-Q> <-C [*configfilesection*]> <-I>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
–h *url*	Specify the dfESP publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window*** Append the following for subscribers: **?snapshot=true** | **false**. Append the following for subscribers if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
–f *fsname*	Specify the subscriber output file, publisher input file, or socket “***host*:*port***". Leave *host* blank to implement a server.
-t binary | **csv** | **xml** | **json** | **syslog** | **hdat**	Specify the file system type. The **syslog** value is valid only for publishers. The **hdat** value is valid only for subscribers.
–c *period*	Specify output file time (in seconds). The active file is closed and a new active file opened with the timestamp appended to the filename
-s *maxfilesize*	Specify the output file volume (in bytes). The active file is closed and a new active file opened with the timestamp appended to the filename.
–d *dateformat*	Specify the date format. The default value = %Y-%m-%d %H:%M:%S
-r *rate*	Specify the requested transmit rate in events per second.
–a *aggrsize*	Specify, in latency mode, statistics for aggregation block size.
–n	Specify latency mode.
-g *gdconfig*	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** 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
–j trace | debug | info | warn | error | fatal | off	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
-b *blocksize*	Set the block size. The default value = 1.
–m *maxevents*	Specify the maximum number of events to publish.
-e	Specify that events are transactional.
-i	Read from growing file.
-p	Buffer all event blocks before publishing them.
–y *logconfigfile*	Specify the log configuration file.
-x *header*	Specify the number of header lines to ignore.
-o *hdat_filename*	Specify the filename to write to Hadoop Distributed File System (HDFS).
-q *hdat_max_datanodes*	Specify the maximum number of data nodes. The default value is the number of live data nodes.
-u *hdfs_blocksize*	Specify the Hadoop Distributed File System (HDFS) block size in MB.
-v *hdfs_numreplicas*	Specify the Hadoop Distributed File System (HDFS) number of replicas. The default value is 1.
-w *hdat_numthreads*	Specify the adapter thread pool size. A value <= 1 means that no data nodes are used.
-z *hdat_max_stringlength*	Specify the fixed size to use for string fields in HDAT records. The value must be a multiple of 8.
-Q	For the publisher, quiesce the project after all events are injected into the source window.
-C [*configfilesection*]	Specifies the name of the section in **/etc/connectors.config** to parse for connection parameters.
-I	When a field in an input CSV event cannot be parsed, insert a null value and continue publishing.
-H *hdatlasrhostport*	Specifies the LASR Analytic Server host and port.
-K *hdatlasrkey*	Specifies the path to TKlasrkey.sh

If you specify Solace or Tervela transports instead of the default native transport, refer to the description of the C++ **`C_dfESPpubsubSetPubsubLib()`** API call for more information about the required client configuration files.

Using the 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*-t *mqtype*
<-q *mqqueuemanager*> <d *dateformat*> <-g *gdconfig*>
<-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*><-C [*configfilesection*]>
<-a *protofile*> <-m *protomsg*>

Publisher usage:

dfesp_mq_adapter -k pub -h *url* —f *mqtopic* -t *mqtype*
—n *mqsubname* —s *mqsubqueue* <-q *mqqueuemanager*>
<-b *blocksize*> <-d *dateformat*> <-e> <-g *gdconfig*>
<-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*> <-C [*configfilesection*]> <-I>
<-a *protofile*> <-m *protomsg*>

Parameter	Definition
`-k`	Specify "sub" for subscriber use and "pub" for publisher use
`-h url`	Specify the dfESP publish and subscribe standard URL in the form "**`dfESP://host:port/project/continuousquery/window`**". Append the following for subscribers: **`?snapshot=true`** | **`false`**. Append the following for subscribers if needed: • **`?collapse=true`** | **`false`** • **`?rmretdel=true`** | **`false`**
`-f mqtopic`	Specify the MQ topic name.
`-t mqtype`	Specify "binary", "CSV", or "JSON".
`-n mqsubname`	Specify the MQ subscription name.
`-s mqsubqueue`	Specify the MQ queue name.
`-q mqqueuemanager`	Specify the MQ queue manager name.
`-b blocksize`	Specify the blocksize. The default value = 1.

Parameter	Definition
-d *dateformat*	Specify the date format. The default value = %Y-%m-%d %H:%M:%S
-e	Specify that events are transactional.
–g *gdconfig*	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y *logconfigfile*	Specify the log configuration file.
-C [*configfilesection***]**	Specify the name of the section in **/etc/connectors.config** to parse for configuration parameters.
–I	When a field in an input CSV event cannot be parsed, insert a null value and continue publishing.
-a *protofile*	Specify the **.proto** file that contains the message used for Google Protocol buffer support.
-m *protomsg*	Specify the message itself in the **.proto** file that is specified by the protofile parameter.

Using the HDAT Reader Adapter

The HDAT Reader adapter resides in **dfx-esp-hdatreader-adapter.jar**, which bundles the Java publisher SAS Event Stream Processing client. The adapter converts each row in an HDAT file into an ESP event and injects event blocks into a source window of an engine. Each event is built with an UPSERT opcode.

The number of fields in the target source window schema must match the number of columns in the HDAT row data. Also, all HDAT column types must be numeric, except for columns that correspond to an ESP field of type UTF8STR. In that case, the column must contain character data.

The source Hadoop Distributed File System (HDFS) and the name of the file within the file system are passed as required parameters to the adapter. The client target platform must define the environment variable **DFESP_HDFS_JARS**. This specifies the location of the Hadoop JAR files.

Usage:

$DFESP_HOME/bin/dfesp_hdat_publisher -u *url* -f *hdfs* -i *inputfile* <-b *blocksize*> <-t> <-g*gdconfigfile*> <-l native | solace |tervela | rabbitmq> <-o severe | warning | info> <-c [*configfilesection*]>

Parameter	Definition
–u *url*	Specify the publish and subscribe standard URL in the form **"dfESP://*host*:*port*/*project*/*continuousquery*/*window*"**.
–f *hdfs*	Specify the target file system, in the form **"hdfs://*host*:*port*"**. Specify the file system that is normally configured in property **fs.defaultFS** in core-site.xml.
–i *inputfile*	Specify the input CSV file, in the form **"/*path*/*filename*.csv"**.
–b *blocksize*	Specify the number of events per event block.
-t	Specify that event blocks are transactional. The default is normal.
– d *dateformat*	Specify the format of **ESP_datetime_t** and **ESP_timestamp_t** fields in events. The default is **"*yyyy*-*MM*-*dd*HH:*mm*:*ss*.*SSS*"**.
- g *gdconfigfile*	Specify the guaranteed delivery configuration file for the client.
–l native | solace | tervela | rabbitmq	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241.
-o severe | warning | info	Specify the application logging level.
-C [*configfilesection*]	Specify the name of the section in file **/etc/javaadapters.config** to parse for configuration parameters.

Using the HDFS (Hadoop Distributed File System) Adapter

The HDFS adapter resides in **dfx-esp-hdfs-adapter.jar**, which bundles the Java publisher and subscriber SAS Event Stream Processing clients. The subscriber client receives event blocks and writes events in CSV format to an HDFS file. The publisher

client reads events in CSV format from an HDFS file and injects event blocks into a source window of an engine.

The target HDFS and the name of the file within the file system are both passed as required parameters to the adapter.

The subscriber client enables you to specify values for HDFS block size and number of replicas. You can configure the subscriber client to periodically write the HDFS file using the optional **periodicity** or **maxfilesize** parameters. If so configured, a timestamp is appended to the filename of each written file.

You can configure the publisher client to read from a growing file. In that case, the publisher runs indefinitely and publishes event blocks whenever the HDFS file size increases.

You must define the **DFESP_HDFS_JARS** environment variable for the client target platform. This variable specifies the location of the Hadoop JAR files.

Subscriber usage:

$DFESP_HOME/bin/dfesp_hdfs_subscriber -u *url* -f *hdfs* -t *outputfile* <-b *hdfsblocksize*> <-n *hdfsnumreplicas*> <-m *maxfilesize*> <-p *periodicity*> <-d *dateformat*> <- g *gdconfigfile*> <-l native | solace | tervela | rabbitmq> <-o severe | warning | info> <-C [*configfilesection*]>

Parameter	Definition
– u *url*	Specify the dfESP subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window***. Append the following for subscribers: **?snapshot=true | false**. Append the following for subscribers if needed: • **?collapse=true | false** • **?rmretdel=true | false**
–f *hdfs*	Specify the target file system, in the form **"hdfs://*host*:*port*"**. Specify the file system normally configured in property **fs.defaultFS** in file core-site.xml.
–t *outputfile*	Specify the output CSV file, in the form **"/*path*/*filename.csv*"**.
–b *hdfsblocksize*	Specify the HDFS block size in MB. The default value is 64MB.
–n *hdfsnumreplicas*	Specify the HDFS number of replicas. The default value is 1.
–m *maxfilesize*	Specify the output file periodicity in bytes.
-p *periodicity*	Specify the output file periodicity in seconds.
– d *dateformat*	Specify the format of **ESP_datetime_t** and **ESP_timestamp_t** fields in events. The default is **"*yyyy-MM-ddHH*:*mm*:*ss.SSS*"**.

Parameter	Definition
–l native | solace | tervela | rabbitmq	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241.
- g *gdconfigfile*	Specify the guaranteed delivery configuration file for the client.
-o severe | warning | info	Specify the application logging level.
-C [*configfilesection*]	Specifies the name of the section in the file **/etc/javaadapters.config** to parse for configuration parameters.

Publisher usage:

$DFESP_HOME/bin/dfesp_hdfs_publisher -u *url* -f *hdfs* –i *inputfile* <-b *blocksize*> <-t> <-d *dateformat*> <- g *gdconfigfile*>
<-l native | solace | tervela | rabbitmq> <-o severe | warning | info> <-C [*configfilesection*]> < -e>

Parameter	Definition
–u *url*	Specify the publish and subscribe standard URL in the form **"dfESP://*host*:*port*/*project*/*continuousquery*/*window*"**.
–f *hdfs*	Specify the target file system, in the form **"hdfs://*host*:*port*"**.
–i *inputfile*	Specify the input CSV file, in the form **"/*path*/*filename*.csv"**.
–b *blocksize*	Specify the number of events per event block.
-t	Specify that event blocks are transactional. The default is normal.
– d *dateformat*	Specify the format of **ESP_datetime_t** and **ESP_timestamp_t** fields in events. The default is **"*yyyy-MM-ddHH:mm:ss.SSS*"**.
- g *gdconfigfile*	Specify the guaranteed delivery configuration file for the client.
–l native | solace | tervela | rabbitmq	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241.

Parameter	Definition
-o severe | **warning** | **info**	Specify the application logging level.
-C [*configfilesection***]**	Specifies the name of the section in file **/etc/javaadapters.config** to parse for configuration parameters.
-e	If a field in an input CSV event cannot be parsed, insert a null value and continue publishing.

Using the Java Message Service (JMS) Adapter

The Java Message Service (JMS) adapter resides in **dfx-esp-jms-adapter.jar**, which bundles the Java publisher and subscriber clients. Both are JMS clients. The subscriber client receives event blocks and is a JMS message producer. The publisher client is a JMS message consumer and injects event blocks into a source window of an engine.

The subscriber client requires a command line parameter that defines the type of JMS message used to contain events. The publisher client consumes the following JMS message types:

- BytesMessage — an Event Streams Processing event block
- TextMessage — an Event Streams Processing event in CSV or XML format
- MapMessage — an Event Stream Processing event with its field names and values mapped to corresponding MapMessage fields

A JMS message in TextMessage format can contain an XML document encoded in a third-party format. You can substitute the corresponding JAR file in the class path in place of dfx-esp-jms-native.jar, or you can use the **-x** switch in the JMS adapter script. Currently, dfx-esp-jms-axeda.jar is the only supported alternative.

When running with an alternative XML format, you must specify the JAXB JAR files in the environment variable DFESP_JAXB_JARS. You can download JAXB from https://jaxb.java.net.

The client target platform must connect to a running JMS broker (or JMS server) . The environment variable **DFESP_JMS_JARS** must specify the location of the JMS broker JAR files. The clients also require a **jndi.properties** file, which you must specify through the **DFESP_JMS_PROPERTIES** environment variable. This properties file specifies the connection factory that is needed to contact the broker and create JMS connections, as well as the destination JMS topic or queue.

You can override the default JNDI names that are looked up by the adapter to find the connection factory and queue or topic. Do this by configuring the **jndidestname** and **jndifactname** adapter parameters. When the destination requires credentials, you can specify these as optional parameters on the adapter command line.

A sample jndi.properties file is included in the **etc** directory of the SAS Event Stream Processing installation.

Subscriber usage:

$DFESP_HOME/bin/dfesp_jms_subscriber -u *url* -m BytesMessage | TextMessage | MapMessage <-i *jmsuserid*> <-p *jmspassword*><-x native | axeda><-d *dateformat*> <- g *gdconfigfile*> <-l native | solace | tervela | rabbitmq> <-o severe | warning | info> <-c [*configfilesection*]> <-f *protofile*> <-r *protomsg*> <-j *jndidestname*> <-a *jndifactname*>

Parameter	Definition
– u *url*	Specify the dfESP subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window***. Append the following for subscribers: **?snapshot=true | false**. Append the following for subscribers if needed: • **?collapse=true | false** • **?rmretdel=true | false**
– d *dateformat*	Specify the format of ESP_datetime_t and ESP_timestamp_t fields in events. The default is "yyyy-MM-dd HH:mm:ss.SSS".
-m BytesMessage | TextMessage | Mapmessage	Specify the JMS message type.
-i *jmsuserid*	Specify the user ID for the JMS destination.
-p *jmspassword*	Specify the password for the JMS destination.
-x native | axeda	Specify the XML format for TextMessage. Specify **native** when the message is in CSV format.
–l native | solace | tervela | rabbitmq	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241.
- g *gdconfigfile*	Specify the guaranteed delivery configuration file for the client.
-o severe | warning | info	Specify the application logging level.
-c [*configfilesection*]	Specify the name of the section of **/etc/javaadapters.config** to parse for configuration parameters.
-f *protofile*	Specify the **.proto** file that contains the message used for Google Protocol buffer support.
-r *protomsg*	Specify the message itself in the **.proto** file that is specified by the protofile parameter.

Parameter	Definition
-j *jndidestname*	Specify the JNDI name to look up for the JMS destination. The default value is **"jmsDestination"**. This option overrides settings in **jndi.properties**.
-a *jndifactname*	Specify the JNDI name to look up for the JMS connection factory. The default value is **"ConnectionFactory"**. This option overrides settings in **jndi.properties**.

Publisher usage:

$DFESP_HOME/bin/dfesp_jms_publisher -u *url* <-b *blocksize*><-i *jmsuserid*> <-p *jmspassword*> <-x native | axeda><-t> <-d *dateformat*> <- g *gdconfigfile*> <-l native | solace | tervela | rabbitmq> <-o severe | warning | info> <-c [*configfilesection*]> <-e> <-f *protofile*> <-r *protomsg*> <-j *jndidestname*> <-a *jndifactname*>

Parameter	Definition
–u *url*	Specify the publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***"
–b *blocksize*	Specify the number of events per event block.
-i *jmsuserid*	Specify the user ID for the JMS destination.
-p *jmspassword*	Specify the password for the JMS destination.
-x native | axeda	Specify the XML format for TextMessage. Specify **native** when the message is in CSV format.
-t	Specify that event blocks are transactional. The default is normal.
– d *dateformat*	Specify the format of ESP_datetime_t and ESP_timestamp_t fields in events. The default is "yyyy-MM-dd HH:mm:ss.SSS".
- g *gdconfigfile*	Specify the guaranteed delivery configuration file for the client.
–l native | solace | tervela | rabbitmq	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241
-o severe | warning | info	Specify the application logging level.
-c [*configfilesection*]	Specify the name of the section in **/etc/javaadapters.config** to parse for configuration parameters.

Parameter	Definition
-e	When a field in an input CSV event cannot be parsed, insert a null value and continue publishing.
-f ***protofile***	Specify the **.proto** file that contains the message used for Google Protocol buffer support.
-r ***protomsg***	Specify the message itself in the **.proto** file that is specified by the protofile parameter.
-j ***jndidestname***	Specify the JNDI name to look up for the JMS destination. The default value is **"jmsDestination"**. This option overrides settings in **jndi.properties**.
-a ***jndifactname***	Specify the JNDI name to look up for the JMS connection factory. The default value is **"ConnectionFactory"**. This option overrides settings in **jndi.properties**.

Using the SAS LASR Analytic Server Adapter

The SAS LASR Analytic Server adapter supports publish and subscribe operations on the SAS LASR Analytic Server. To run this adapter in a UNIX environment requires the installation of an SSH client.

Note: Do not run this adapter in a Microsoft Windows environment.

Subscriber usage:

dfesp_lasr_adapter -k sub -h *url1<, url2,...urlN>* -H *lasrurl* -t *table* -X *tklasrkey* <-s *tabschema* | —a *obstoanalyze*> <-d *dformat*> <-A *commit*> <-S true | false> <-b *blocksize*> <-f *hdfspath* | —F *hdfsfullpath*> <-B *hdfsblock*> <-c *periodcity*> <-r true | false> <-n true | false> <-l *loglevel*> <-z *bufsize*><-R *numreplicas*> <-C [*configfilesection*]> <-g *gdconfigfile*> <-L native | solace | tervela | rabbitmq>

Publisher usage:

dfesp_lasr_adapter -k pub -h *url1<, url2,...urlN>* -H *lasrurl* -t *table* -X *tklasrkeypath* <-S true | false> <-d *dformat*> <-e true | false> <-E true | false> <-b *blocksize*> <-l *loglevel*> <-q *action* | —Q *action*> <-z *bufsize*><-C [*configfilesection*]> <-g *gdconfigfile*> <-L native | solace | tervela | rabbitmq>

Parameter	Description
-k sub | **pub**	Specify subscribe or publish.

Parameter	Description
-h ***url***	Specify the publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window***. You must append the following for subscribers: **?snapshot=true** | **false**. As an option, append **?collapse=true** | **false** for subscribers. *Note:* You can specify multiple URLs.
-H ***lasrurl***	Specify the SAS LASR Analytic Server URL in the form **"*host*:*port*"**.
-t ***table***	Specify the full name of the LASR table.
-X ***tklasrkeypath***	Specifies the path to **tklasrkey.sh**
-s ***tabschema***	Specify the explicit SAS LASR Analytic Server schema in the form "*rowname1*:*sastype1*:*sasformat1*:*label1*,...*rownameN*:*sastypeN*:*sasformatN*:*labelN*. *Note:* When you omit the *sasformat*, no SAS format is applied to the row. When you omit the *label*, no label is applied to the row. If you previously specified **-a**, this option is ignored.
-a ***obstoanalyze***	Specify the number of observations to analyze in order to define the output SAS LASR Analytic Server structure. The default value is 4096. When you specify **-s**, this option is ignored.
-d ***dformat***	Specify the datetime format. The default is **yyyy-MM-dd HH:mm:ss**.
-A ***acommit***	Specify the number of observations to commit to the server. A value < 0 specifies the time in seconds between auto-commit operations. A value > 0 specifies the number of records that, after reached, forces an auto-commit operation. A value of 0 specifies no auto-commit operation. The default value is 0.
-S true | **false**	Specify whether to observe strict SAS LASR Analytic Server adapter schema.
-b ***blocksize***	For a subscriber, specify the buffer size (number of observations) to be flushed to the server. For a publisher, specify the event block size to be published to a source window. The default is 1.
-f ***hdfspath***	Specify the path to the SAS Hadoop file system where the LASR table should be saved. If you specify **-F**, this option is ignored.

Parameter	Description
-F ***hdfsfullpath***	Specify the full path to the SAS Hadoop file system where the LASR table should be saved. If you specify **-f**, this option is ignored.
-B ***hdfsblock***	Specify the SAS Hadoop file system block size. The value must be a multiple of 512 bytes. The default is 64KB.
-c ***periodicity***	For a subscriber, specify the interval in seconds after which to save data to the SAS Hadoop file system. The default value is 0. *Note:* The file is marked with **_YYYYMMDD_HHMMSS**.
-r true | **false**	Specify whether to replace the file on the SAS Hadoop file system.
-n true | **false**	Specify whether to create-recreate a LASR table.
-l ***loglevel***	Specify the Java standard logging level. Valid values are **OFF** | **SEVERE** | **WARNING** | **INFO** | **CONFIG** | **FINE** | **FINER** | **FINEST** | **ALL**. The default value is **INFO**.
-z ***bufsize***	Specify the socket buffer size.
-R ***numreplicas***	Specify the number of replicas to create for the SAS Hadoop file system.
-C [*configfilesection***]**	Specify the name of a section in **/etc/javaadapters.config** to parse for configuration parameters.
-e true | **false**	Specify whether event blocks are transactional. If **false**, event blocks are normal. The default value is **false**.
-E true | **false**	For a publisher, specify whether the adapter fetches all data from the LASR table. When **false**, data is cached.
-q ***action***	For a publisher, specify the action to get results from the LASR table. For example, specify **–q "fetch field1 / to=100 where field2=2"**.
-Q ***action***	For a publisher, specify the action to get cached results from the LASR table. For example, specify **–q "fetch field1 / to=100 where field2=2"**.
-L native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default native transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241.

Parameter	Description
-g *gdconfig*	Specify the guaranteed delivery configuration file.

Using the PI Adapter

The PI Adapter supports publish and subscribe operations against a PI Asset Framework (PI) server. You must install the PI AF Client from OSISoft in order to use the adapter.

Subscriber usage:

dfesp_pi_adapter -k sub -h *url* -s *afelement*<-t> <-p *pisystem*><-d *afdatabase*> <-r *afrootelement*><-a *afattribute*><-g *gdconfig*><-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal> <-y *logconfigfile*> <-C [*configfilesection*]>

Publisher usage:

dfesp_pi_adapter -k pub-h *url* -s *afelement*<-t> <-p *pisystem*><-d *afdatabase*><-r *afrootelement*> <-a *afattribute*><-c><-b *blocksize*> <-e><-g *gdconfig*> <-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal> <-y *logconfigfile*> <-C [*configfilesection*]>

Parameter	Definition
-k	Specify **sub** for subscriber use and **pub** for publisher use
-h *url*	Specify the dfESP publish and subscribe standard URL in the form **dfESP://*host*:*port*/*project*/*continuousquery*/*window***. Append the following for subscribers: **?snapshot=true | false**. Append the following for subscribers if needed: **?collapse=true | false**.
-s *afelement*	Specify the AF element or element template name. Wildcards are supported.
-t	Specify that *afelement* is the name of an element template.
-p *pisystem*	Specify the PI system.
–d *afdatabase*	Specify the AF database
–r *afrootelement*	Specify a root element in the AF hierarchy from which to search for *afelement*.
-a *afattribute*	Specify an attribute in *afelement* and ignore other attributes there.
-b *blocksize*	Specify the block size. The default value is 1.

Parameter	Definition
-c ***archivetimestamp***	Specify that when connecting to the AF server, retrieve all archived values from the specified timestamp onwards.
—e	Specify that events are transactional.
—g ***gdconfig***	Specify the guaranteed delivery configuration file.
—l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
-j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
—y ***logconfigfile***	Specify the log configuration file.
-C [*configfilesection***]**	Specify the name of the section of **/etc/connectors.config** to parse for configuration parameters.

Using the Rabbit MQ Adapter

The Rabbit MQ adapter supports publish and subscribe operations on a Rabbit MQ server. You must install the **rabbitmq-c** client run-time libraries to use the adapter.

Subscriber usage:

dfesp_rmq_adapter -k sub -h *url* -u *rmquserid* -p *rmqpassword* -s *rmqhost*
-r *rmqport* -v *rmqexchange* -t *rmqtopic* -z binary | csv | json
-o *urlhostport* -n *numbufferedmsgs* <-d *dateformat*> <-x > <-f *protofile*> <-m *protomsg*>
<-g *gdconfig*> <-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-C *configfilesection*> <-w never | once | pereventblock> <-c>

Publisher usage:

dfesp_rmq_adapter -k pub -h *url* -u *rmquserid* -p *rmqpassword* -s *rmqhost* -r *rmqport*
-v *rmqexchange* -t *rmqtopic* -z binary | csv | json
-o *urlhostport* < -x > <-d *dateformat*> <-f *protofile*> <-m *protomsg*>
<-g *gdconfig* > <-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-C *configfilesection*> <-b *blocksize*> <-e > <-I > <-q *buspersistencequeue*>

Parameter	Definition
-k	Specify **sub** for subscriber or **pub** for publisher.

Parameter	Definition
– h ***url***	Specify the dfESP subscribe standard URL in the form **dfESP://***host*:*port*/***project*/*continuousquery*/*window***. Append the following for subscribers: **?snapshot=true | false**. Append the following for subscribers if needed: • **?collapse=true | false** • **?rmretdel=true | false**
-u ***rmquserid***	Specify the Rabbit MQ user name.
-p ***rmqpassword***	Specify the Rabbit MQ password.
-s ***rmqhost***	Specify the Rabbit MQ host.
-r ***rmqport***	Specify the Rabbit MQ port.
-v ***rmqexchange***	Specify the Rabbit MQ exchange.
-t ***rmqtopic***	Specify the Rabbit MQ routing key.
–z binary | csv | json	Specify the message format.
-o ***urlhostport***	Specify the *host*:*port* field in the metadata topic to which the connector subscribes.
-x	Use durable queues and persistent messages.
-d ***dateformat***	Specify the format of **ESP_datetime_t** and **ESP_timestamp_t** fields in events. The default is **"yyyy-MM-dd HH:mm:ss.SSS"**.
-f ***protofile***	Specify the **.proto** file to be used for Google protocol buffer support.
-m ***protomsg***	Specify the message itself in the **.proto** file that is specified by the **protofile** parameter.
-g ***gdconfig***	Specify the guaranteed delivery configuration file.
-l native | solace | tervela | rabbitmq	Specify the publish/subscribe transport. The default is **native**.
-j trace | debug | info | warn | error | fatal | off	Specify the logging level. The default is **warn**.

Parameter	Definition
-y *logconfigfile*	Specify the logging configuration file. By default, there is none.
-C *configfilesection*	Specify the name of the section in **/etc/connectors.config** to parse for configuration parameters, in the form **[*section*]**.
-n *numbufferedmsgs*	Specify the maximum number of messages buffered by a standby subscriber connector.
-w never | once | pereventblock	Specify when to pass the window schema to every subscriber callback. The default value is **never**. When **rmqtype=CSV**, prepend output CSV data with the window's serialized schema.
-b *blocksize*	Specify event block size. The default is 1.
-c	Specify to use the client message ID instead of an internal message ID to synchronize traffic during failover.
-e	Specify that events are transactional.
-l	Insert a null value into fields that fail CSV parsing, and then continue.
-q*buspersistencequeue*	Specify the queue name used by a persistent publisher.

Using the REST Subscriber Adapter

The REST adapter supports subscribe operations to generate HTTP POST requests to a configured REST service. For each subscribed event, the adapter formats a JSON string that uses all fields of the event. It then forwards the JSON through an HTTP POST to the REST service that is configured in the adapter parameter ***resturl***. You can also configure the HTTP Content-Type and the number of retries when an HTTP POST fails.

The HTTP response can be forwarded to an unrelated source window. This requires building an event from the JSON formatted response and forwarding it to the ESP URL that is configured in the ***esprespurl*** adapter parameter. When the JSON response includes an array, a new event is built and published for each array entry, with non-array fields unchanged.

The response event fields are appended to the fields in the original subscribed event. Thus, you must be careful to ensure that the beginning fields in the schema of the source window in ***esprespurl*** exactly match the complete schema of the subscribed window. Also, the response fields must follow the beginning fields.

Each HTTP request-response action is run in a separate adapter thread. In this way, adapter memory usage does not grow with queued subscribed events waiting synchronously for the previous request-response to complete.

Usage:

dfesp_rest_subscriber -u *url* -r *resturl* -t *httpcontenttype* <-e *esprespurl*> <-p *httpretries*> <-d *dateformat*> <- g *gdconfigfile*> <-l native | solace | tervela> <-o severe | warning | info> <-c [*configfilesection*]>

Parameter	Definition
-h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*contquery*/*window***".
-r *resturl*	Specify the URL of the target REST service.
-t *httpcontenttype*	Specify the value of the Content-Type string used in the HTTP post.
-e *esprespurl*	Specify the publish/subscribe standard URL to which responses should be published.
-p *httpretries*	Specify the number of times to retry a failed HTTP post. The default value is 0.
-d *dateformat*	Specify the format of **ESP_datetime_T** and **ESP_timestamp_t** fields in events. The default value is "yyyy-MM-dd HH:mm:ss.SSS".
-l native | solace | tervela	Specify the transport type. If you specify **solace** or **tervela** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
- g *gdconfigfile*	Specify the guaranteed delivery configuration file for the client.
-o severe | warning | info	Specify the application logging level.
-C [*configfilesection*]	Specifies the name of the section in **/etc/javaadapters.config** to parse for connection parameters.

Using the SAS Data Set Adapter

The SAS Data Set adapter resides in dfx-esp-dataset-adapter.jar, which bundles the Java publisher and subscriber SAS Event Stream Processing clients.

The adapter uses SAS Java Database Connectivity (JDBC) to connect to a SAS Workspace Server. This SAS Workspace Server manages reading and writing of the data set. The SAS data set name and SAS Workspace Server user credentials are passed as required parameters to the adapter.

The SAS JDBC requires the **`log4j.jar`** for logging. Configure the value of the DFESP_LOG4J_JAR environment variable to be the path to **`log4j.jar`**.

The user password must be passed in encrypted form. The encrypted version of the password can be generated by using OpenSSL, which must be installed on your system. Use the following command on the console to invoke OpenSSL to display your encrypted *password*:

echo "*password*" | **openssl** enc -e -aes-256–cbc -a -salt
-pass "pass:SASespDSadapterUsedByUser="*userid*"

The subscriber client receives event blocks and appends corresponding rows to the SAS data set. It also supports Update and Delete operations on the data set. The data set is created by the Workspace Server on its local file system, so the data set name must comply with these SAS naming rules:

- The length of the names can be up to 32 characters.
- Names must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore. Subsequent characters can be letters of the Latin alphabet, numerals, or underscores.
- Names cannot contain blanks or special characters except for the underscore.
- Names can contain mixed-case letters. SAS internally converts the member name to uppercase.

You can explicitly specify the data set schema using the **`-s`** option. You can use the **`-a`** switch to specify a number of received events to analyze and to extract an appropriate row size for the data set before creating data set rows. One or the other switch must be present.

You can also configure the subscriber client to periodically write a SAS data set using the optional **`periodicity`** or **`maxnumrows`** parameters. If so configured, a timestamp is appended to the filename of each written file. Be aware that Update and Delete operations are not supported if **`periodicity`** or **`maxnumrows`** is configured, because the referenced row might not be present in the currently open data set.

If you have configured a subscriber client with multiple ESP URLs, events from multiple windows in multiple models are aggregated into a single output data set. In this case, each subscribed window must have the same schema. Also, the order in which rows are appended to the data set is not guaranteed.

The publisher client reads rows from the SAS data set, converts them into ESP events with **`opcode = upsert,`** and injects event blocks into a source window of an engine. If you configure a publisher client with multiple ESP URLs, each generated event block is injected into multiple ESP source windows. Each source window must have the same schema.

Subscriber usage:

$DFESP_HOME/bin/dfesp_dataset_adapter -k sub -h *url1…urlN*
-f *dsname* -d *wssurl* -u *username* -x *password* <-s *dsschema* | —a *obstoanalyze*>
<-p *periodicity*> <-m *maxnumrows*> <-l *loglevel*> <-g *gdconfigN*> <-z *bufsize*>
<—C *configfilesection*> <-g *gdconfigfile*> <-L native | solace | tervela | rabbitmq>

Publisher usage:

$DFESP_HOME/bin/dfesp_dataset_adapter -k pub -h *url1…urlN*
-f *dsname* -d *wssurl* -u *username* -x *password* <-e true | false><-b *blocksize*>
<-l *loglevel*> <-g *gdconfigN*> <-z *bufsize*> <—C *configfilesection*><-q *sqlquery*>
<-g *gdconfigfile*> <-L native | solace | tervela | rabbitmq>

Parameter	Definition
-k pub | **sub**	Specify a publisher or a subscriber.
-h *url*	Specify one or more standard dfESP URLs in the following form: **dfESP://*host*:*port*/*project*/*continuousquery*/*window*** Append the following to the URL for subscribers if needed: **?snapshot=true** | **false** Append the following for subscribers if needed: **?collapse=true** | **false**
-f *dsname*	Specify the subscriber output file or publisher input file. Specify a full name with the extension.
-d *wssurl*	Specify the SAS Workspace Server connection URL in the form **"*host*:*port*"**.
-u *username*	Specify the SAS Workspace Server user name.
-x *password*	Specify the SAS Workspace Server password in encrypted form.
-s *dsschema*	Specify the output data set schema in the following form: **"*rowname1*:*sastype1*:*sasformat1*:*label1*,...,*rownameN*:*sastypeN*:*sasformatN*:*labelN*** *Note:* If you omit *sasformatX*, then no SAS format is applied to the row. If you omit *labelX*, then no label is applied to the row. If you had previously specified the **-a** option, this option is ignored.
—a *obstoanalyze*	Specify the number of observations to analyze to define the output SAS data set structure. Default value is 4096. If you had previously specified the **-s** option, this option is ignored.
—m *maxnumrows*	Specify the output file periodicity in number of rows. Invalid if data is not insert-only.
—p *periodicity*	Specify the output file periodicity in seconds. Invalid if data is not insert-only.
— b *blocksize*	For a subscriber, specify the number of event blocks to be appended to the data set at once. For a publisher, specify the event block size to be published to the ESP source window. Default value is 1.

Parameter	Definition
–e true | false	When true, event blocks are transactional. Otherwise, they are normal. The default is false.
-l ***loglevel***	Specify the Java standard logging level. Use one of the following values: **OFF** | **SEVERE** | **WARNING** | **INFO** | **CONFIG** | **FINE** | **FINER** | **FINEST** | **ALL**. Default value is **INFO**.
-C [*configfilesection***]**	Specify the name of the section in file **/etc/javaadapters.config** to parse for configuration parameters.
-z ***bufsize***	Specify the socket buffer size.
-q ***sqlquery***	Specify an SQL query to get SAS data set content. For example: **"select * from** ***dataset*****"**. *Note:* Use the *dataset* specified for the **-f** option without an extension.
-g ***gdconfig***	Specify the guaranteed delivery configuration file.
-L native | solace | tervela | rabbitmq	Specify the transport type. When you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default native transport, use the required client configuration files specified in "Using Alternative Transport Libraries for Java Clients" on page 241.

Using the SMTP Subscriber Adapter

The SMTP Subscriber Adapter is subscriber only. Publishing to a source window is not supported. This adapter forwards subscribed event blocks or single events as e-mail messages to a configured SMTP server, with the event data in the message body encoded in CSV format.

Subscriber use:

dfesp_smtp_adapter-h *url*-m *smtpserver* -u *sourceaddress*
-d *destaddress* <-p> <-g *gdconfig*> <-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*><-C [*configfilesection*]>

Parameter	Definition
-h ***url***	Specify the dfESP subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window*?snapshot=true** | **false**. You can append the following if needed: • **?collapse=true** | **false** • **?rmretdel=true** | **false**
-p	Specify that each e-mail contains a single event instead of a complete event block containing one or more events.
-g	Specify the guaranteed delivery configuration file.
–l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
-y ***logconfigfile***	Specify the log configuration file.
-C [*configfilesection***]**	Specify the name of the section in **/etc/connectors**.config to parse for connection parameters.
-m ***smtpserver***	Specify the SMTP server name.
-u ***sourceaddress***	Specify the source email address.
-d ***destaddress***	Specify the destination email address

Using the Sniffer Publisher Adapter

The sniffer adapter supports publish operations of events that are created from packets captured from a local network interface in promiscuous mode. You must install the **libpcap** run-time libraries in order to use this adapter.

Publisher use:

dfesp_sniffer_adapter -h *url* -i *interface* -p radius | http -f *packetfields* <-g *gdconfig*> <-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*> <-C *configfilesection*> <-b *blocksize*> <-e > <-t> <-d *vendorID*> <-v *vendorType*>

Parameter	Definition
-h ***url***	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*contquery*/*window***".
-i ***interface***	Specify the name of the network interface on the local machine from which to capture packets.
-p radius | **http**	Specify the protocol type of packets to be captured.
-f ***packetfields***	Specify a list of fields to be extracted from captured packets.
-g ***gdconfig***	Specify the name of the guaranteed delivery configuration file.
-l native | **solace** | **tervela** | **rabbitmq**	Specify the publish/subscribe transport. The default is **native**.
-j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Specify the logging level. The default is **warn**.
-y ***logconfigfile***	Specify the name of the logging configuration file. The default is none.
-C ***configfilesection***	Specify the name of section in **/etc/connectors.config** to parse for configuration parameters, in the form **"[*section*]"**
-b ***blocksize***	Specify the event block size. The default value is 1.
-e	Specify that events are transactional.
-t	Specify to append an **ESP_TIMESTAMP** field to each published event.
-d ***vendorID***	Specify the vendor ID field to match when capturing the Attribute-Specific field in a Vendor-Specific attribute in a Radius Accounting-Request packet.
-v ***vendorType***	Specify the vendor type filed to match when capturing the Attribute-Specific field in a Vendor-Specific attribute in a Radius Accounting-Request packet.

Using the Solace Systems Adapter

The Solace Systems adapter supports publish and subscribe operations on a hardware-based Solace fabric. You must install the Solace run-time libraries to use the adapter.

Subscriber usage:

dfesp_sol_adapter -k sub -h *url* -u *soluserid*
—p *solpassword* —v *solvpn* —t *soltopic*
—o *urlhostport* -n *numbufferedmsgs* <-b > <-g *gdconfig*>
<-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-f *protofile*> <-m *protomsg*> <-C [*configfilesection*]>

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 | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-f *protofile*> <-m *protomsg*> <-C [*configfilesection*]>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
-h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***”. Append the following for subscribers: **?snapshot= true | false**. Append the following for subscribers if needed: • **?collapse= true | false** • **?rmretdel=true | false**
-u *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 | rabbitmq	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** 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
–j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y ***logconfigfile***	Specify the log configuration file.
-b	Use Solace Guaranteed Messaging. By default, Solace Direct Messaging is used.
–q ***buspersistencequeue***	Specify the queue name used by Solace Guaranteed Messaging publisher.
-f ***protofile***	Specify the **.proto** file that contains the message used for Google Protocol buffer support.
-m ***protomsg***	Specify the message itself in the **.proto** file that is specified by the **protofile** parameter.
-C [*configfilesection***]**	Specify the name of the section in **/etc/connectors.config** to parse for configuration parameters.

Using the Teradata Subscriber Adapter

The Teradata adapter supports subscribe operations against a Teradata server, using the Teradata Parallel Transporter for improved performance. You must install the Teradata Tools and Utilities (TTU) to use the adapter.

Note: A separate database adapter uses generic DataDirect ODBC drivers for the Teradata platform. For more information, see "Using the Database Adapter" on page 298.

Usage:

dfesp_tdata_adapter -h *url* -d stream | update | load -u *username* -x *userpwd* -t *tablename* -s *servername* -a *maxsessions* -n *minsessions* -i true | false <-b *batchperiod*> <-q *stage1table*> <-r *stage2table*> <-c *connectstring*> <-z *tracelevel*> <-g *gdconfig*> <-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*> <-C [*configfilesection*]>

Parameter	Description
-h *url*	Specify the dfESP publish and subscribe standard URL in the form **dfESP://***host***:***port***/***project***/***continuousquery***/***window***?snapshot=true** | **false**". Append **?collapse=true** | **false** for subscribers if needed. Append **?rmretdel=true** | **false** for subscribers if needed.
-d stream | **update** | **load**	Specify the Teradata operator. For more information about these operators, see the documentation provided with the Teradata Tools and Utilities.
-d *username*	Specify the Teradata user name.
-x *userpwd*	Specify the Teradata user password.
-t *tablename*	Specify the name of the target table on the Teradata server.
-s *servername*	Specify the name of the target Teradata server.
-a *maxsessions*	Specify the maximum number of sessions on the Teradata server. A Teradata session is a logical connection between an application and Teradata Database. It allows an application to send requests to and receive responses from Teradata Database. A session is established when Teradata Database accepts the user name and password of a user.
-n *minsessions*	Specify the minimum number of sessions on the Teradata server.
-i	Specify that the subscribed window contains insert-only data.
-b *batchperiod*	Specify the batch period in seconds. This parameter is required when **-d** specifies the update operator.
-q *stage1table*	Specify the name of the first staging table on the Teradata server. This parameter is required when **-d** specifies the load operator.
-r *stage2table*	Specify the name of the second staging table on the Teradata server. This parameter is required when **-d** specifies the load operator.
-c *connectstring*	Specify the connect string to be used by the ODBC driver to access the staging and target tables on the Teradata server. Use the form **"DSN=***dsnname***;uid=***userid***;pwd=***password*. This parameter is required when **-d load**.

Parameter	Description
-z ***tracelevel***	Specify the trace level for Teradata messages written to the trace file in the current working directory. The trace file is named *operator*1.txt. Default value is 1 (TD_OFF). Other valid values are: 2 (TD_OPER), 3 (TD_OPER_CLI), 4 (TD_OPER_OPCOMMON), 5 (TD_OPER_SPECIAL), 6 (TD_OPER_ALL), 7 (TD_GENERAL), and 8 (TD_ROW).
-g ***gdconfig***	Specify the guaranteed delivery configuration file.
-l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default native transport, use the required client configuration files specified in the description of the **C++ C_dfESPpubsubSetPubsubLib()** API call.
-j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. Use the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call or in the engine **initialize()** call.
-y ***logconfigfile***	Specify the log configuration file.
-C [*configfilesection***]**	Specify the name of the section in **/etc/connectors.config** to parse for configuration parameters.

Using the Tervela Data Fabric Adapter

The Tervela adapter supports publish and subscribe operations on a hardware-based or software-based Tervela fabric. You must install the Tervela run-time libraries to use the adapter.

Subscriber usage:

dfesp_tva_adapter -k sub -h *url* —u *tvauserid*
—p *tvapassword* —t *tvaprimarytmx* —f *tvatopic*
—c *tvaclientname* -m *tvamaxoutstand* —b *numbufferedmsgs*
-o *urlhostport* <-s *tvasecondarytmx*> <-l *tvalogfile*>
<-w *tvapubbwlimit*> <-r *tvapubrate*><-e *tvapubmsgexp*>
<-g *gdconfig*> <-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off> <-y *logconfigfile*> <-C [*configfilesection*]>
<-a *protofile*> <-m *protomsg*>

Publisher usage:

dfesp_tva_adapter -k pub -h *url* —u *tvauserid*
—p *tvapassword* —t *tvaprimarytmx* —f *tvatopic*
—c *tvaclientname* -n *tvasubname*
-o *urlhostport* <-s *tvasecondarytmx*> <-l *tvalogfile*>
<-g *gdconfig*> <-l native | solace | tervela | rabbitmq>
<-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-C [*configfilesection*]>
<-a *protofile*> <-m *protomsg*>

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use
-h *url*	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***”. Append the following for subscribers: **?snapshot=true | false** Append the following for subscribers if needed: • **?collapse=true | false** • **?rmretdel=true | false**
-u *tvauserid*	Specify the Tervela user name.
-p *tvapassword*	Specify the Tervela password.
-t *tvaprimarytmx*	Specify the Tervela primary TMX.
-f *tvatopic*	Specify the Tervela topic.
-c *tvaclientname*	Specify the Tervela client name.
-m *tvamaxoutstand*	Specify the Tervela maximum number of unacknowledged messages.
-b *numbufferedmsgs*	Specify the maximum number of messages buffered by a standby subscriber connector.
-o *urlhostport*	Specify the *host*:*port* string sent in connector metadata message
—s *tvasecondarytmx*	Specify the Tervela secondary TMX.
—i*tvalogfile*	Specify the Tervela log file. The default is syslog.
—w *tvapubbwlimit*	Specify the Tervela maximum bandwidth of published data (Mbps). The default value is 100.
-r *tvapubrate*	Specify the Tervela publish rate (Kbps). The default value is 30.

Parameter	Definition
—e ***tvapubmsgexp***	Tervela maximum time to cache published messages (seconds); the default value is 1
—g ***gdconfig***	Specify the guaranteed delivery configuration file.
—l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
—j trace | **debug** | **info** | **warn** | **error** | **fatal** | **off**	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
—y ***logconfigfile***	Specify the log configuration file.
-C [*configfilesection***]**	Specify the section in **/etc/connectors.config** to parse for configuration parameters.
-a ***protofile***	Specify the **.proto** file that contains the message used for Google Protocol buffer support.
-m ***protomsg***	Specify the message itself in the **.proto** file that is specified by the protofile parameter.

Using the Tibco Rendezvous (RV) Adapter

The Tibco RV adapter supports publish and subscribe operations using a Tibco RV daemon. You must install the Tibco RV run-time libraries to use the adapter.

Subscriber usage:

dfesp_tibrv_adapter -k sub -h *url* -f *tibrvsubject* —t *tibrvtype*
-s *tibrvservice*<-n *tibrvnetwork*> <-m *tibrvdaemon*>
<-d *dateformat*> <-g *gdconfig*>
<-l native | solace | tervela | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-C [*configfilesection*]>
<-a *protofile*> <-b *protomsg*>

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 | rabbitmq> <-j trace | debug | info | warn | error | fatal | off>
<-y *logconfigfile*> <-C [*configfilesection*]> <-I>
<-a *protofile*> <-b *protomsg*>

Table 17.1 *Parameter Definitions*

Parameter	Definition
-k	Specify “sub” for subscriber use and “pub” for publisher use.
-h ***url***	Specify the dfESP publish and subscribe standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window*"**.** Append the following for subscribers: **"?snapshot= true | false "**. Append the following for subscribers if needed: • **?collapse= true | false** • **?rmretdel= true | false**
-f ***tibrvsubject***	Specify the Tibco RV subject.
-t ***tibrvtype***	Specify “binary”, “CSV”, or “JSON”.
-s ***tibrvservice***	Specify the Tibco RV service.
-n ***tibrvnetwork***	Specify the Tibco RV network.
-m ***tibrvdaemon***	Specify the Tibco RV daemon.
-b ***blocksize***	Specify the block size. The default value is 1.
–d ***dateformat***	Specify the date format. The default is %Y-%m-%d %H:%M:%S.
–e	Specify that events are transactional.
–g ***gdconfig***	Specify the guaranteed delivery configuration file.
–l native | solace | tervela | rabbitmq	Specify the transport type. If you specify **solace**, **tervela**, or **rabbitmq** transports instead of the default **native** transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
–j trace | debug | info | warn | error | fatal | off	Set the logging level for the adapter. This is the same range of logging levels that you can set in the **C_dfESPpubsubInit()** publish/subscribe API call and in the engine **initialize()** call.
–y ***logconfigfile***	Specify the log configuration file.
-C [*configfilesection***]**	Specify the name of the section of **/etc/connectors.config** to parse for configuration parameters.

Parameter	Definition
-I	When a field in an input CSV event cannot be parsed, insert a null value and continue publishing.
-a ***protofile***	Specify the **.proto** file that contains the message used for Google Protocol buffer support.
-b ***protomsg***	Specify the message itself in the **.proto** file that is specified by the protofile parameter.

Using the Twitter Publisher Adapter

The Twitter adapter is publisher only. It consumes Twitter streams and injects event blocks into source windows of an engine. The Twitter adapter provides the following capabilities:

- subscribes to Twitter streams and receives a continuous flow of tweets as soon as they are tweeted.
- filters the incoming streams using the following standard features provided by Twitter:
  - follows specific users.
  - tracks specific keywords.
  - filters specific geographical locations.
  - filters specific languages.
  - receives random generated tweets.
  - receives only RT tweets. This requires specific Twitter account authorization.
  - receives only tweets that include links. This requires specific Twitter account authorization.
  - receives full “fire hose.” This requires specific Twitter account authorization.
- publishes received tweets to a source window for processing inside a model.

Usage:

dfesp_twitter_publisher -u *url* -m sample | filter | links | firehose | retweet <-f *followlist*> <-k *tracklist*> < -p *locations*> <-a *languages*> <-C *prevcount*> < -b *blocksize*> <-t> <-d *dateformat*>< -g *gdconfigfile*> <-c [*configfilesection*]> <-T *twitterpropfile*> < -l native | solace | tervela | rabbitmq> <-o severe | warning | info>

Parameter	Definition
–u ***url***	Specify the publish standard URL in the form "**dfESP://*host*:*port*/*project*/*continuousquery*/*window***".

Parameter	Definition
`-m sample | filter | links | firehose | retweet`	Specify the method type for subscribing to Twitter streams. See the table that follows on page 333.
`-f` ***`followlist`***	Specify the list of user IDs to follow with the filter method in the form “user1,user2,user3”. The default access level allows up to 400 follower user IDs. See https://dev.twitter.com/streaming/overview/request-parameters for more details.
`-k` ***`tracklist`***	Specify the list of keywords to track with the filter method in the form “keyword1,keyword2,keyword3”. The default access level allows up to 200 track keywords. See Streaming API request parameters for more information.
`-p` ***`locations`***	Specify the list of locations to track with the filter method in the form of a comma-separated list of longitude,latitude pairs. Each pair specifies a set of bounding boxes with the southwest corner of the bounding box coming first. Sample:“-122.75,36.8,-121.75,37.8” for San Francisco. Note that bounding boxes do not act as filters for other filter parameters. The default access level allows up to 10 1-degree location boxes. See Streaming API request parameters for more information.
`-a` ***`languages`***	Specify a comma-separated list of BCP 47 language identifiers to return only Tweets that have been detected as being written in the specified languages. For example, connecting with “en,fr” streams only Tweets detected to be in the English or French language.
`-C` ***`prevcount`***	Used with filter, link, or retweet method. Indicates the number of previous statuses to stream before transitioning to the live stream. Default is 0. The supplied value can be an integer from 1 to 150000 or from -1 to -150000. If a positive number is specified, the stream transitions to live values after the backfill has been delivered to the client. When a negative number is specified, the stream disconnects after the backfill has been delivered to the client, which might be useful for debugging. This parameter requires elevated access to use. See Streaming API request parameters for more information.

Parameter	Definition
-b ***blocksize***	Specify the number of events per event block.
-t	Specify that event blocks are transactional. The default is normal.
-d ***dateformat***	Specify the format of **ESP_datetime_t** and **ESP_timestamp_t** fields in events. The default is "**yyyy-MM-dd HH:mm:ss.SSS**".
-c [*configfilesection***]**	Name of section in **/etc/javaadapters.config** to parse for configuration parameters.
-T ***twitterpropfile***	Name of file in **/etc** to parse for Twitter configuration parameters. Default is **twitterpublisher.properties**.
-g ***gdconfigfile***	Specify the guaranteed delivery configuration file for the client.
-l native | **solace** | **tervela** | **rabbitmq**	Specify the transport type. If you specify Solace, Tervela, or RabbitMQ transports instead of the default native transport, use the required client configuration files specified in the description of the C++ **C_dfESPpubsubSetPubsubLib()** API call.
-o severe | **warning** | **info**	Specify the application logging level.

Twitter sets limitations to access streams and uses some methods depending on the user-account level of access:

Access Method	Limitations
sample	Starts listening on random sample of all public statuses. The default access level provides a small proportion of the **firehose**. The "Gardenhose" access level provides a proportion more suitable for data mining and research applications that need a statistically significant sample.

Access Method	Limitations
`filter`	Starts consuming public statuses that match one or more filter predicates. At least one predicate parameter, follow, locations, or track must be specified. Multiple parameters can be specified that enable most clients to use a single connection to the Streaming API. Placing long parameters in the URL can cause the request to be rejected for excessive URL length. The default access level allows up to 200 track keywords, 400 follower user IDs and 10 1-degree location boxes. Increased access levels enable the following: • 80,000 follower user IDs ("shadow" role) • 400,000 follower user IDs ("birddog" role) • 10,000 track keywords ("restricted track" role) • 200,000 track keywords ("partner track" role) • 200 10-degree location boxes ("locRestricted" role) Increased track access levels also pass a higher proportion of statuses before limiting the stream. For more information about specifying parameters, see Streaming API request parameters on the Twitter website.
`firehose`	Starts listening on all public statuses. Available only to Twitter approved parties and requires a signed agreement to access.
`retweet`	Starts listening on all retweets. The retweet stream is not a generally available resource. Few applications require this level of access. Creative use of a combination of other resources and various access levels can satisfy nearly every application use case.
`link`	Starts listening on all public statuses containing links. Available only to Twitter approved parties and requires a signed agreement to access.

When you install this adapter, be mindful of the following:

- The `twitterpublisher.properties` file is required. SAS Event Stream Processing provides a sample one in `$DFESP_HOME/etc`.
- You must download the files `twitter4j-core-4.0.2.jar` and `twitter4j-stream-4.0.2.jar` from twitter4j.org and copy them to `$DFESP_HOME/lib`.

To configure this adapter:

1. Set up and authorize a Twitter user account to be used by the Twitter adapter to receive tweets.
   a. Sign in to Twitter.
   b. After signing in, go to http://dev.twitter.com.
   c. Select **Manage Your Apps**. This directs you to the Application Management page.
   d. Click **Create New App**.
   e. Complete the application details. Click **Create your Twitter application**.
   f. You are prompted to review and adjust your application's settings. Click **Keys and Access Tokens**.
   g. Click **Create my access token**. Twitter generates your access tokens.

2. Enter the generated access tokens in the configuration file **`twitterpublisher.properties`** as follows:

```
twitter4j.oauth.consumerKey=generated_consumer_key_API_key
twitter4j.oauth.consumerSecret=generated_consumer_key_API_Secret_key
twitter4j.oauth.accessToken=generated_access_token
twitter4j.oauth.accessTokenSecret=generated_access_token_secret
```

3. Save and close **`twitterpublisher.properties`**.
4. If you need to add other settings such as proxy servers, API stream URLs, or trace settings, seehttp://twitter4j.org/en/configuration.html#Streaming and edit the **`twitterpublisher.properties`** file accordingly.

Chapter 18

Visualizing Event Streams

Overview to Event Visualization 337
Using Streamviewer 337

Overview to Event Visualization

You can visualize event streams two ways:

- SAS Event Stream Processing Studio
- Streamviewer, which is provided with SAS Event Stream Processing

You can use SAS Event Stream Processing Studio to subscribe to windows. See Chapter 4, “Using SAS Event Stream Processing Studio,”.

Using Streamviewer

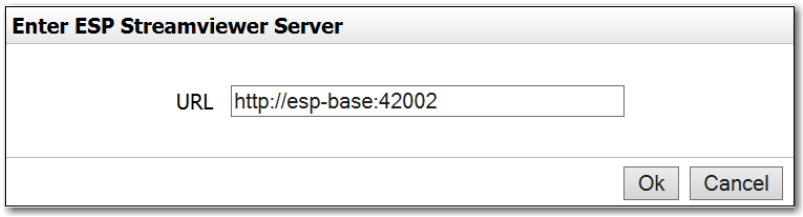
Streamviewer enables you to subscribe to a running event stream processing model and display the events streaming through it. Each event is displayed as a row in a table. Each row of the table is keyed by the schema key of the corresponding window.

You can find Streamviewer in **`$DFESP_HOME/share/tools/streamviewer`**. To run it, do one of the following:

- Copy **`$DFESP_HOME/share/tools/streamviewer`** to the directory on a web server where web applications run. Open a browser and enter **`http://`*`yourserver`*`:`*`yourport`*`/streamviewer/streamviewer.html`**.
- Copy the contents of **`$DFESP_HOME/share/tools/streamviewer`** to a local computer system. Navigate to that directory and double-click **`streamviewer.html`**.

After Streamviewer is running, enter a URL that points to an event stream processing publish/subscribe HTTP provider. Use the following form for the URL: **`http://`**

provider:port, where *provider* is an event stream processing publish/subscribe HTTP provider and *port* is a publish/subscribe port.



The provider can reside in a process within a C++ application or in the XML server.

Specify the following on a command line to start an event stream processing publish/subscribe HTTP provider: **dfesp_xml_server -http-pubsub [*port*]** where *port* is the publish/subscribe port that you have enabled.

Alternatively, specify the publish/subscribe HTTP provider within an XML model:

```
<http-servers>
    <http-pubsub-server port='[http pubsub port]'>
  </http-servers>
```

If you want to run the XML server with HTTP publish/subscribe enabled, you must associate the server with a publish/subscribe port.

To view events streaming through a C++ application, send **-h [pubsub http *port*]** to the **dfESPengine::initialize()** call in your application.

Note: The Streamviewer UI uses HTML5 local storage to store the specified URL and configuration items. Therefore, you must use a browser that supports HTML5.

You subscribe to a running event stream processing model in one of three modes:

Update
: The opcode of an event is used to add, modify, or delete the event's corresponding row in the table. When you create an Update table, it grabs the current snapshot of the model and populates the table with the snapshot's events.

Streaming
: An event is appended to the end of the table and the event's opcode is displayed. A streaming table displays a limited number of rows. When the maximum is reached, the oldest events are removed from the beginning.

  To change the number of rows streamed through the table, click on the user interface (UI). The default number of rows is 500.

Streaming with Snapshot
: Behaves the same as Streaming mode except that the subscription is populated with a current snapshot of the window elements.

Note: Streaming a large number of rows can affect the performance of the UI.

Streamviewer is supported on the Google Chrome browser.

Chapter 19

Enabling Encryption on Socket Connections

Overview to Enabling Encryption 339
Understanding SSL Certificate Requirements 340
Understanding the SSL Handshake Process 340

Overview to Enabling Encryption

You can enable encryption on TCP/IP connections within an engine. Specifically, you can encrypt the following:

- connections that are created by a client using the C or Java publish/subscribe API to connect to an event stream processing server.
- connections that are created by an adapter connecting to an event stream processing server.
- connections that are created by a file and socket connector or adapter that acts as a socket client or server. In this case, the TCP peer can be another file and socket connector or adapter, or a third-party socket application.

  Note: Suppose a file and socket connector or adapter connects to a SAS LASR Analytic Server. Suppose further that the server runs on a Hadoop Distributed File System (HDFS) name node in order to access SAS HDAT files. In this case, encryption is not supported.

You must meet three requirements in order to enable encryption:

- The OpenSSL libraries must be installed on the machines running the client and server.
- The **`DFESP_SSLPATH`** environment variable must be defined with the path to the OpenSSL shared object or DLL.

  Note: For a Java publish/subscribe client or adapter, the value of **`DFESP_SSLPATH`** is not important. SSL is supported natively in Java. Nevertheless, **`DFESP_SSLPATH`** must be defined.
- The proper SSL certificates must be installed on the client and server. If encryption is not enabled through **`DFESP_SSLPATH`**, the installation runs successfully without OpenSSL being installed.

Understanding SSL Certificate Requirements

Consider the following SSL certificate requirements in order to enable encryption:

- The required SSL certificates differ for client and server. You must copy the needed certificate files into **$DFESP_HOME/etc**.
- The server is an engine with publish/subscribe enabled or a file and socket connector or adapter running as a server. The server requires the following files in **$DFESP_HOME/etc**:
  - **server.pem**

    This file must contain a concatenation of a certificate and private key.
  - **key.passphrase** (only when the key is password-protected)
- A client is a C or Java publish/subscribe client, an adapter, or a file and socket connector or adapter running as a client. The client requires the **ca.pem** file in **$DFESP_HOME/etc**. This file must contain a certificate used to verify the received server side certificate.

  For certificates signed by a Certificate Authority (CA), the certificate of the signer must be present. For self-signed certificates, the certificate can be a copy of the certificate in **server.pem**.

Production traffic must use only certificates that are signed by a CA.

The encryption method uses the default cipher used by the OpenSSL installation on the local machine.

Understanding the SSL Handshake Process

When **DFESP_SSLPATH** is defined but SSL certificates cannot be found or the OpenSSL library cannot be loaded, a fatal error is logged at start-up. If there are no start-up errors, the publish/subscribe server indicates whether SSL is enabled or disabled through an INFO level log message logged when the server starts up.

A client that is enabled for SSL logs a handshake-status INFO-level message when it initiates its connection. When a client or server is negotiating SSL and its peer is not, or when the SSL handshake itself fails, the connection fails and an error message is logged.

Chapter 20

Enabling Guaranteed Delivery

Overview to Guaranteed Delivery 341
Guaranteed Delivery Success Scenario 343
Guaranteed Delivery Failure Scenarios 344
Additions to the Publish/Subscribe API for Guaranteed Delivery 344
Configuration File Contents 345

Overview to Guaranteed Delivery

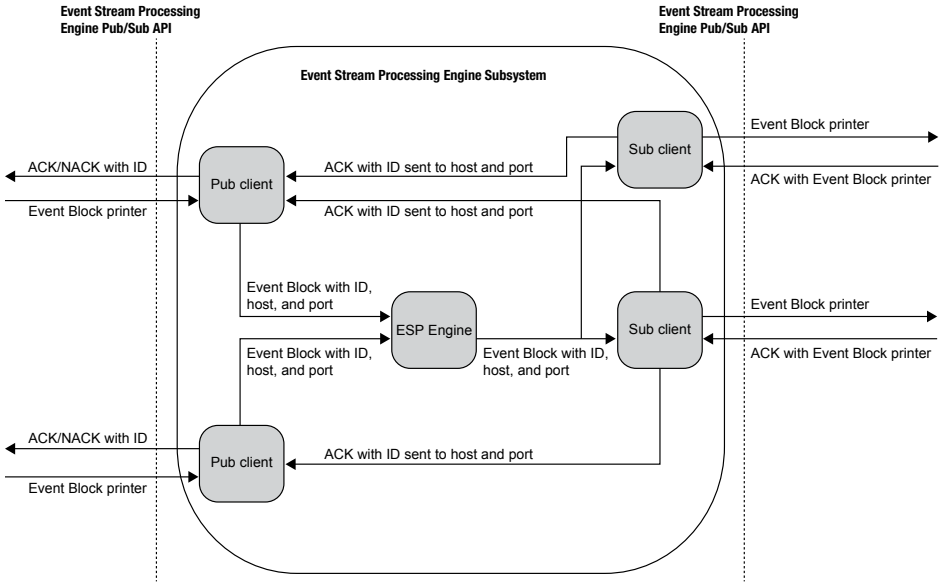
Both the Java and C publish and subscribe (pub/sub) APIs support guaranteed delivery between a single publisher and multiple subscribers. Guaranteed delivery assumes a model where each event block that is published into a source window generates exactly one event block in a subscribed window. This one block in, one block out principle must hold for all published event blocks. The guaranteed delivery acknowledgment mechanism is not aware of 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 or 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 a TCP connection to each publisher on the list. This connection is then used only to transport acknowledgments specific to this publisher/subsciber pair. 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.

Figure 20.1 *Guaranteed Delivery Data Flow Diagram*



Please note the following:

- Publishers and subscribers that do not use the guaranteed-delivery-enabled API functions are implicitly guaranteed delivery disabled.
- Guaranteed delivery subscribers can be mixed with non-guaranteed delivery subscribers.
- A guaranteed delivery-enabled publisher might wait to begin publishing until a READY callback has been received. This indicates that its configured number of subscribers have established their acknowledgment connections back to the publisher.
- Event blocks received by a guaranteed-delivery-enabled subscriber as a result of a snapshot generated by the engine are not acknowledged.
- Under certain conditions, subscribers receive duplicate event blocks. These conditions include the following:
    - A publisher begins publishing before all related subscribers have started. Any started subscriber can receive duplicate event blocks until the number of started subscribers reaches the number of required acknowledgments passed by the publisher.
    - A guaranteed delivery-enabled subscriber disconnects while the publisher is publishing. This triggers the same scenario described previously.
    - A slow subscriber causes event blocks to time-out, which triggers a negative acknowledgment 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 acknowledgment connection, it retries at a configurable rate up to a configurable maximum number of retries.

- Suppose that a guaranteed delivery-enabled publisher injects an event block that contains an ID, and that the ID is present in the publish client's not acknowledged-ID list. In that case, the inject call is rejected by the publish client. The ID is cleared from the list when the publish client passes it to the ACK/NACK callback function of the new publisher.

Guaranteed Delivery Success Scenario

In the context of guaranteed delivery, the publisher and subscriber are customer applications that are the endpoints in the data flow. The subscribe and publish clients are event stream processing code that implements the publish/subscribe API calls made by the publisher and subscriber.

The flow of a guaranteed delivery success scenario is as follows:

1. The publisher passes an event block to the publish client, where the ID field in the event block has been set by the publisher. The publish client fills in the host-port field, adds the ID to its unacknowledged ID list, and injects it to the engine.
2. The event block is processed by the 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 acknowledgment connection that matches the host or port in the event block, completely bypassing the engine.
6. Upon receipt of the acknowledgment, the publish client increments the number of acknowledgments received for this event block. If that number has reached the threshold passed to the publish client at start-up, the publish client invokes the new guaranteed delivery callback with parameters acknowledged and ID. It removes the ID from the list of unacknowledged IDs.

Guaranteed Delivery Failure Scenarios

There are three failure scenarios for guaranteed delivery flows:

Scenario	Description
Event Block Time-out	• An event block-specific timer expires on a guaranteed-delivery-enabled publish client, and the number of acknowledgments received for this event block is below the required threshold. • The publish client invokes the new guaranteed delivery callback with parameters **`NACK`** and **`ID`**. No further retransmission or other attempted recovery by the publish client or subscribe client is undertaken for this event block. The publisher most likely backs out this event block and resends. • The publish client removes the ID from the list of unacknowledged IDs.
Invalid Guaranteed Delivery Acknowledged Connect Attempt	• A guaranteed-delivery-enabled publish client receives a connect attempt on its guaranteed delivery acknowledged server but the number of required client connections has already been met. • The publish client refuses the connection and logs an error message. • For any subsequent event blocks received by the guaranteed delivery-enabled subscribe client, an error message is logged.
Invalid Event Block ID	• A guaranteed-delivery-enabled publisher injects an event block that contains an ID already present in the publish client's unacknowledged ID list. • The inject call is rejected by the publish client and an error message is logged.

Additions to the Publish/Subscribe API for Guaranteed Delivery

The publish/subscribe API provides the following methods to implement guaranteed delivery sessions:

- **`C_dfESPGDpublisherStart()`**
- **`C_dfESPGDsubscriberStart()`**
- **`C_dfESPGDsubscriberAck()`**
- **`C_dfESPGDpublisherCB_func()`**
- **`C_dfESPGDpublisherGetID()`**

For more information, see "Functions for the C Publish/Subscribe API" on page 225. For publish/subscribe operations without a guaranteed delivery version of the function, call the standard publish/subscribe API function.

Configuration File Contents

The publish client and subscribe client reads a configuration file at start-up to get customer-specific configuration information for guaranteed delivery. The format of both of these files is as follows.

Guaranteed Delivery-enabled Publisher Configuration File Contents	
Local port number for guaranteed delivery acknowledgment connection server.	
Time-out value for generating negative acknowledgments, in seconds.	
Number of received acknowledgments required within time-out period to generate positive instead of negative acknowledgments.	
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.	
Acknowledgment connection retry interval, in seconds.	
Acknowledgment connection maximum number of retry attempts.	
File Format:	**GDsub_pub=<host:port>** **GDsub_retryInt=<interval>** **GDsub_maxretries=<max>**

Chapter 21

Implementing 1+N-Way Failover

Overview to 1+N-Way Failover **347**

Topic Naming **350**
Overview to Topic Naming 350
Rabbit MQ and Solace 350
Tervela 351

Failover Mechanisms **351**
Overview to Failover Mechanisms 351
Determining ESP Active/Standby State (RabbitMQ) 351
Determining ESP Active/Standby State (Solace) 351
Determining ESP Active/Standby State (Tervela) 353
New ESP Active Actions on Failover (Rabbit MQ) 353
New ESP Active Actions on Failover (Solace) 353
New ESP Active Actions on Failover (Tervela) 354

Restoring Failed Active ESP State after Restart **355**

Using ESP Persist/Restore **355**

Message Sequence Numbers **355**

Metadata Exchanges (Rabbit MQ and Solace) **356**

Metadata Exchanges (Tervela) **356**

Required Software Components **357**

Required Client Configuration **357**

Required Appliance Configuration (Rabbit MQ) **357**

Required Appliance Configuration (Solace) **357**

Required Appliance Configuration (Tervela) **358**

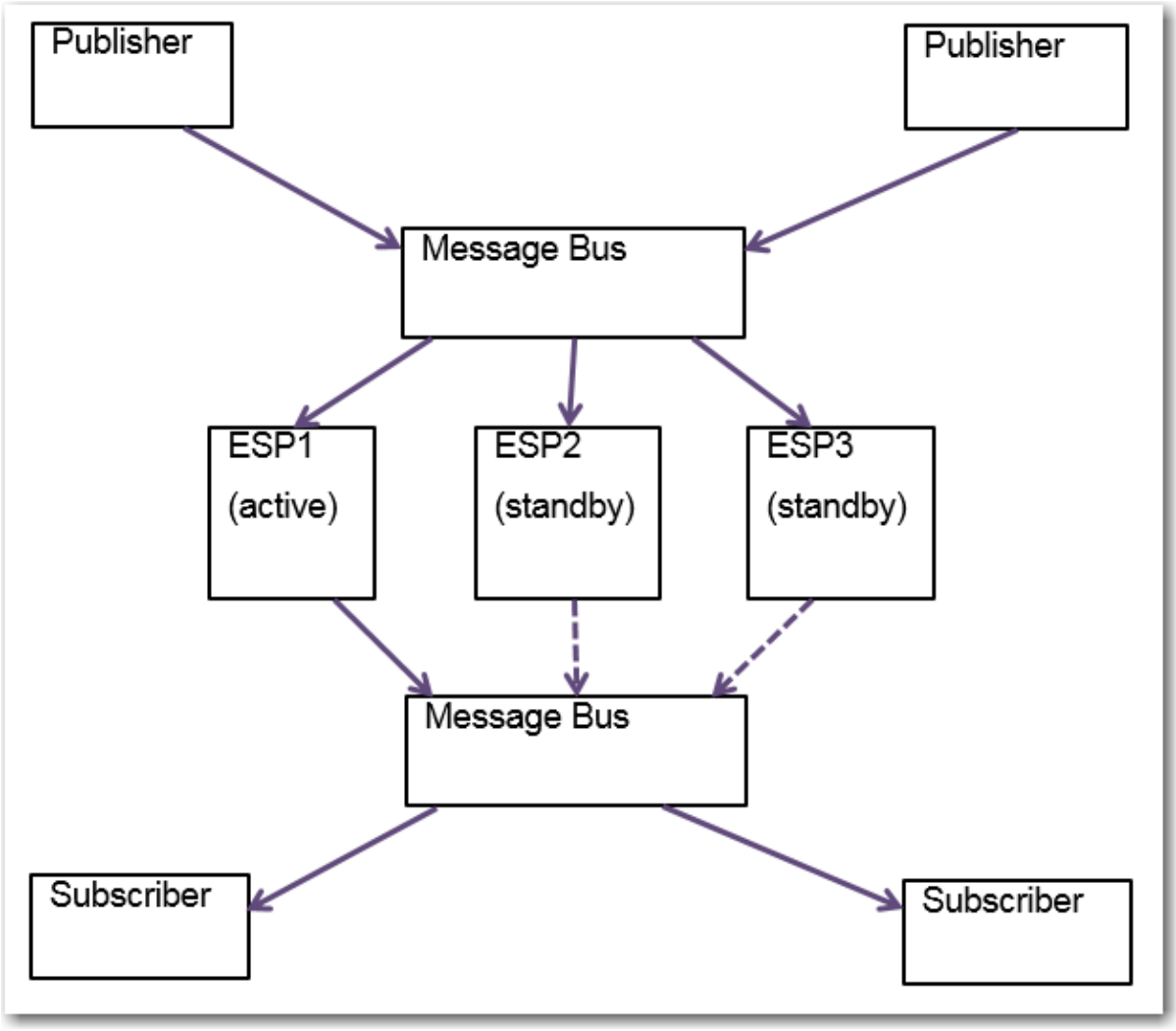
Overview to 1+N-Way Failover

SAS Event Stream Processing can use message buses 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 message bus, 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. 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 a message bus with respect to another message bus in a redundant pair. The terms 1+N, failover, cluster, and combinations of these terms are used interchangeably.

The following diagram shows how an engine integrates with message buses 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 21.1 *Engine Integration with Message Buses*



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 message bus. This distinction is depicted with dotted arrows. The event stream processing message bus connector for subscribe services is connected to the fabric. It does not actually sending event blocks to the message bus 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 message bus.

Publishers and subscribers can continue to use the ESP API even when they are subscribing or publishing through the message bus for failover.

The following transport options are supported by the publish/subscribe API and in adapter configuration. These options are supported so that failover can be introduced to an existing implementation without reengineering the subscribers and publishers:

- native
- Rabbit MQ
- Solace
- Tervela

However, when you use the message bus for publish/subscribe, the event stream processing API uses the message bus 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 Rabbit MQ, Solace, or Tervela connectors to communicate with the message bus. Like client publishers and subscribers, they are effectively subscribers and publishers. They subscribe to the message bus for messages from the publishers. They publish to the message bus so that it can publish messages to the subscribers.

These message buses support using direct (that is, non-persistent) or persistent messaging modes. Rabbit MQ connectors implement non-persistence by declaring non-durable auto-delete queues. They implement persistence by declaring durable non-auto-delete queues. The durable queues require explicit message acknowledgment, which the connector does not do. Messages are read but not consumed from the queue when they are not acknowledged. Solace fabrics can use either direct or persistent messaging. The Tervela connector requires that Tervela fabrics use persistent messaging for all publish/subscribe communication between publishers, ESPs, and subscribers.

Enabling persistent messaging on the message bus 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 message 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 covered depends on the configuration of the message bus.
- At the start of the day, the message bus 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 event stream processor's 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.

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.

Rabbit MQ and Solace

The topic name format used on Rabbit MQ and Solace appliances is as follows: ***`host:port/project/contquery/window/direction`***, where *direction* takes the value "I" or "O". Because all this information is present in a client URL, it is easy for clients to determine the correct appliance topic. ESP appliance connectors use their configured "**`urlhostport`**" parameter to derive the "*host*:*port*" section of the topic name, and the rest of the information is known by the connector.

Tervela

The topic name format used on Tervela appliances is as follows: "**`SAS.ENGINES.`*`engine.project.contquery.window.direction`***", where *direction* takes the value "IN" or "OUT". ESP appliance connectors know this information, so it is easy for them to determine the correct appliance topic.

Clients must be able to map the "*host*:*port*" section of the received URL to the engine section of the topic name. This mapping is obtained by the client by subscribing to a special topic named **`SAS.META.`*`host:port`***.. The ESP appliance connectors use their configured "**`urlhostport`**" parameter to build this topic name,. They publish a metadata message to the topic that includes the "*host*:*port*" to engine mapping. Only after receiving this message can clients send or receive event block data. ESP appliance connectors automatically send this message when the ESP model is started.

Failover Mechanisms

Overview to Failover Mechanisms

If the active engine (ESP) in a failover cluster fails, the standby ESP appliance connectors are notified. Then one of them becomes the new active ESP. The fabric tells the new active connector the ID of the last message that it received on the window-specific "out" topic. The new active connector begins sending data on that "out" topic with ID + 1.

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

Note: Failover support is unavailable when Google Protocol buffer support or JSON messaging is enabled.

Determining ESP Active/Standby State (RabbitMQ)

You must have installed the presence-exchange plug-in on the Rabbit MQ server. All ESP subscribers declare a Rabbit MQ exchange of type **`x-presence`**. The exchange is named after the configured exchange name with **`_failoverpresence`** appended. Then subscribers bind to a queue to both send and receive notifications of bind and unbind actions by all ESP peers.

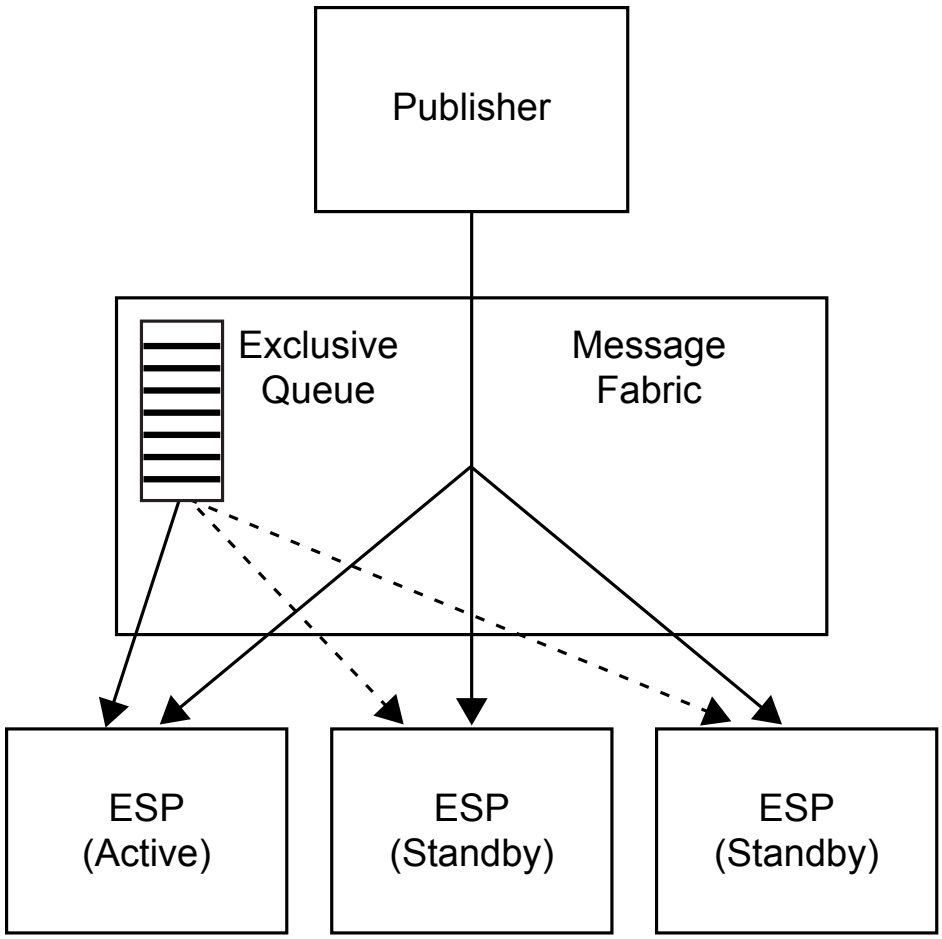
All ESPs receive send and receive notifications in the same order. Therefore, they maintain the same ordered list of present ESPs (that is, those that are bound). The first ESP in the list is always the active ESP. When a notification is received, an ESP compares its current active/standby state to its position in the list and updates its active/standby state when 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 21.2 *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 (Rabbit MQ)

When a subscriber connector starts in standby state, it creates a queue that is bound to the out topic that is used by the currently active connector. The subscriber consumes and discards all messages received on this queue, except for the last one received. When its state changes from standby to active, the subscriber extracts the message ID from the last received message, deletes its **`receive`** queue, and starts publishing starting with the following message:

```
ID = last message ID + 1
```

The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. It discards older messages from the queue.

New ESP Active Actions on Failover (Solace)

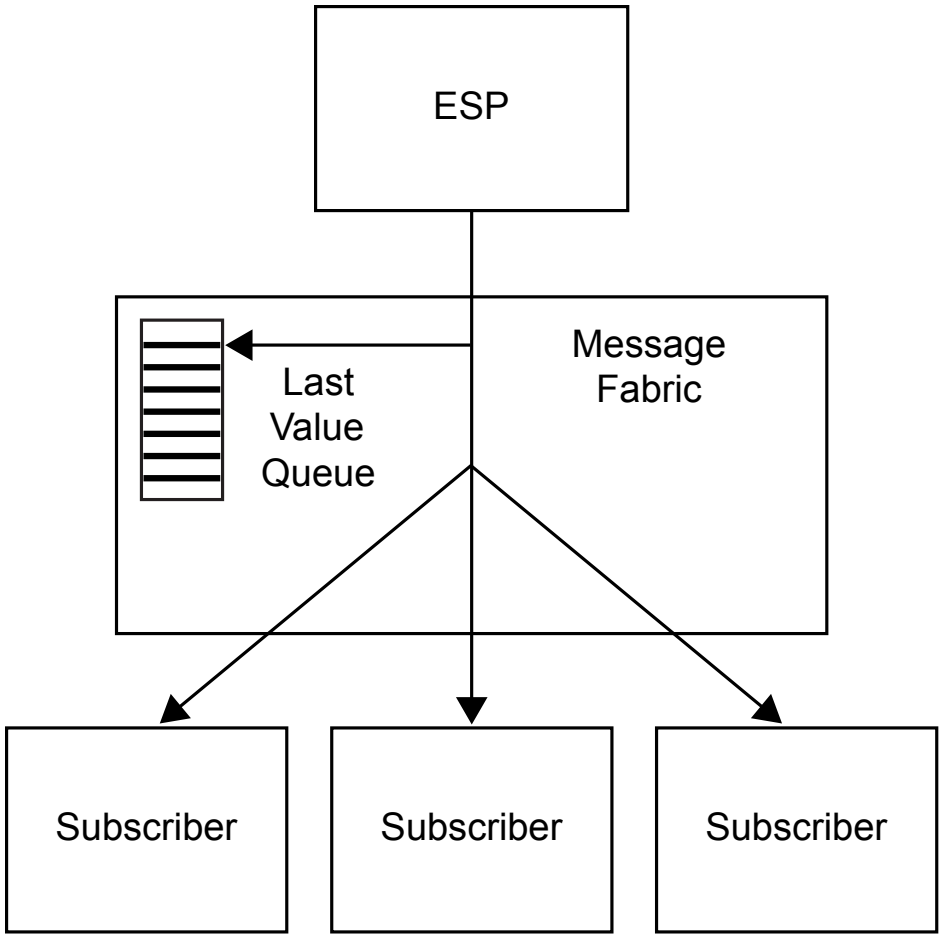
The newly active engine (ESP) determines, from the message bus, the last message published by the previously active ESP for the relevant window. To assist in this process, guaranteed messaging Last Value Queues (LVQs) are used.

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 21.3 *Last Value Queues*



New ESP Active Actions on Failover (Tervela)

Active Tervela appliance connectors own a cluster-wide Tervela GD context with a name that matches the configured “**`tvaclientname`**” parameter. This parameter must be identical for all subscribe appliance connectors in the same failover cluster. When a connector goes active because of a failover, it takes over the existing GD context. This allows it to query the context for the ID of the last successfully published message, and this message ID is saved.

The connector then starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. Alternatively, it can ignore newly received messages until the one with ID = saved message ID + 1 is received.

Restoring Failed Active ESP State after Restart

When you manually bring a failed active ESP back online, it is made available as a standby when another ESP in the cluster is currently active. If the message bus is operating in “direct” mode, persisted messages on the topic do not replay. The standby ESP remains out-of-sync with other ESPs with injected event blocks. When the message bus 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 message bus configuration and disk resources. In many cases, the data persisted might not be enough to cover 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 a Rabbit MQ server, this is achieved by having the connector use a well-known queue name. Only a single queue exists, and the first ESP to consume the persist request performs the persist action. On Solace appliances, this is achieved by setting Deliver-To-One on the persist message to the metadata topic. On the Tervela Data Fabric this is achieved by sending the persist message to an inbox owned by only one ESP in the failover cluster.

The persist data is always written to disk. The target path for the persist data is specified in the client persist API method. Any client that requests persists of an ESP in a specific failover cluster should specify the same path. This path can point to shared disk, so successive persists do not have to be executed by the same ESP in the failover cluster.

The other requirement is that the model must execute a restore on boot so that a rebooted standby ESP can synchronize its state using the last persisted snapshot. On start-up, appliance connectors always get the message ID of the last event block that was restored. If the restore failed or was not requested, the connector gets 0. This message ID is compared to those of all messages received through replay by a persistence-enabled appliance. Any duplicate messages are ignored.

Message Sequence Numbers

The message IDs that are used to synchronize ESP failovers are generated by the ESP engine. They are inserted into an event block when that event block is injected into the

model. This ID is a 64-bit integer that is unique within the scope of its project/query/window, and therefore unique for the connector. When redundant ESP engines receive identical input, this ID is guaranteed to be identical for an event block that is generated by different engines in a failover cluster.

The message IDs used to synchronize a rebooted ESP with published event blocks are generated by the inject method of the Rabbit MQ, Solace, or Tervela publisher client API. They are inserted into the event block when the event block is published into the appliance by the client. This ID is a 64-bit integer that is incremented for each event block published by the client.

Metadata Exchanges (Rabbit MQ and Solace)

The Rabbit MQ and Solace publish/subscribe API handles the **`C_dfESPpubsubQueryMeta()`** and **`C_dfESPpubsubPersistModel()`** methods as follows:

- The 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 (for Solace) or a well-known queue name with multiple consumers (for Rabbit MQ). This is 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, Tervela, or RabbitMQ 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, Tervela, or RabbitMQ publish/subscribe API provided with SAS Event Stream Processing. For C or C++ applications, the Solace, Tervela, or RabbitMQ transport option is requested by calling **`C_dfESPpubsubSetPubsubLib()`** before calling **`C_dfESPpubsubInit()`**. For Java applications, the Solace, Tervela, or RabbitMQ transport option is invoked by inserting dfx-esp-solace-api.jar, dfx-esp-tervela-api.jar, or dfx-esp-rabbitmq-api.jar into the classpath in front of dfx-esp-api.jar.
- You must install the Solace, Tervela, or RabbitMQ run-time libraries on platforms that host running instances of the connectors and clients. SAS Event Stream Processing 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.

A RabbitMQ client application requires a client configuration file named **`rabbitmq.cfg`** in the current directory to provide Rabbit MQ connectivity parameters.

See the documentation of the **`C_dfESPpubsubSetPubsubLib()`** publish/subscribe API function for details about the contents of these configuration files.

Required Appliance Configuration (Rabbit MQ)

You must install the presence-exchange plug-in in order to use the Rabbit MQ server in a 1+N Way Failover topology. You can download the plug-in from https://github.com/tonyg/presence-exchange.

Required Appliance Configuration (Solace)

A Solace appliance used in a 1+N Way Failover topology requires the following 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 *buspersistencequeue* 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 a 1+N Way Failover topology requires the following configuration at a minimum:

- A client user name and password to match the connector's **`tvauserid`** and **`tvapassword`** configuration parameters.
- The inbound and outbound topic strings and associated schema. (See topic string formats described previously.)
- Publish or subscribe entitlement rights associated with a client user name described previously.

Chapter 22

Using Design Patterns

Overview to Design Patterns **359**
Design Pattern That Links a Stateless Model with a Stateful Model **360**
Controlling Pattern Window Matches **361**
Augmenting Incoming Events with Rolling Statistics **362**

Overview to Design Patterns

A design pattern is a reusable solution to a common problem within a specific context of software design. This chapter provides design patterns to common problems encountered in event stream processing application programming.

Event stream processing models can be stateless, stateful, or mixed. The type of model that you choose affects how you design it. The combinations of windows that you use in your design pattern should enable fast and efficient event stream processing. One challenge when designing a mixed model is to identify sections that must be stateful and those that can be stateless, and then connecting them properly.

A stateless model is one where the indexes on all windows have the type **`pi_EMPTY`**. In this case, events are not retained in any window, and are essentially transformed and passed through. These models exhibit fast performance and use very little memory. They are well-suited to tasks where the inputs are inserts and when simple filtering, computation, text context analysis, or pattern matching are the only operations required.

A stateful model is one that uses windows with index types that store data, usually **`pi_RBTREE`** or **`pi_HASH`**. These models can fully process events with Insert, Update, or Delete opcodes. A stateful model facilitates complex relational operations such as joins and aggregations. Because events are retained in indexes, whenever all events are Inserts only, windows grow unbounded in memory. Thus, stateful models must process a mix of Inserts, Updates, and Deletes in order to remain bounded in memory.

The mix of opcodes can occur in one of two ways:

- The data source and input events have bounded key cardinality. That is, there are a fixed number of unique keys in the input stream (such as customer IDs). There can be many updates to these keys provided that the key cardinality is finite.
- A retention policy is enforced for the data flowing in, where the amount of data is limited by time or event count. The data is then automatically deleted from the system by the generation of internal retention delete events.

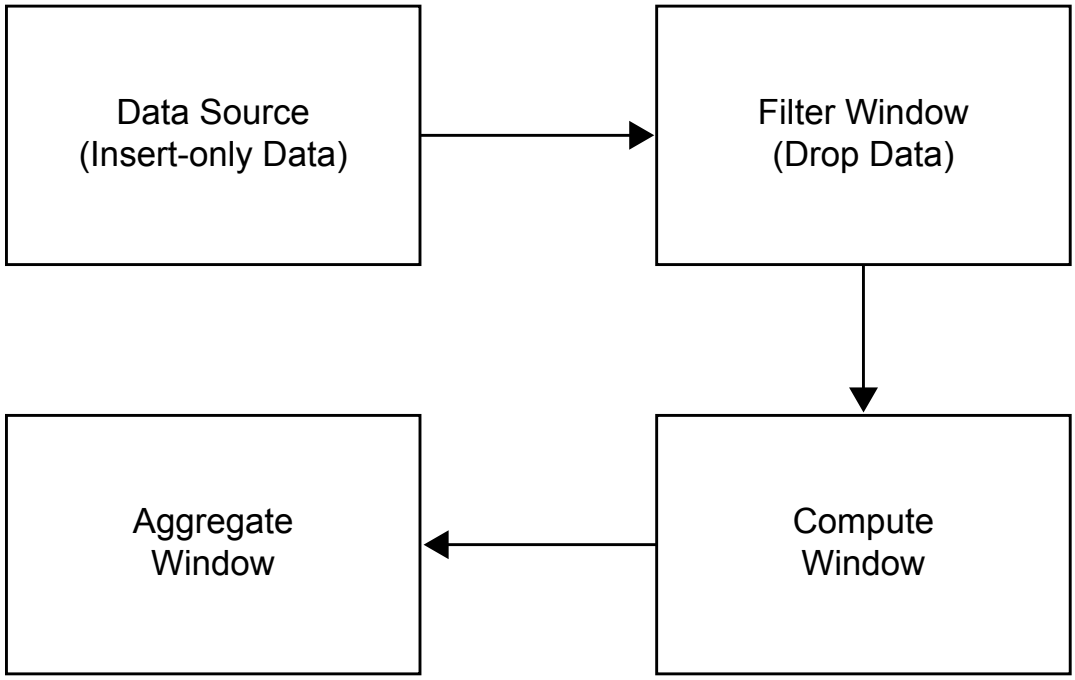
A mixed model has stateless and stateful parts. Often it is possible to separate the parts into a stateless front end and a stateful back end.

Design Pattern That Links a Stateless Model with a Stateful Model

To control memory growth in a mixed model, link the stateless and stateful parts with copy windows that enforce retention policies. Use this design pattern when you have insert-only data that can be pre-processed in a stateless way. Pre-process the data before you flow it into a section of the model that requires stateful processing (using joins, aggregations, or both).

For example, consider the following model:

Figure 22.1 *Event Stream Processing Model with Insert-Only Data*



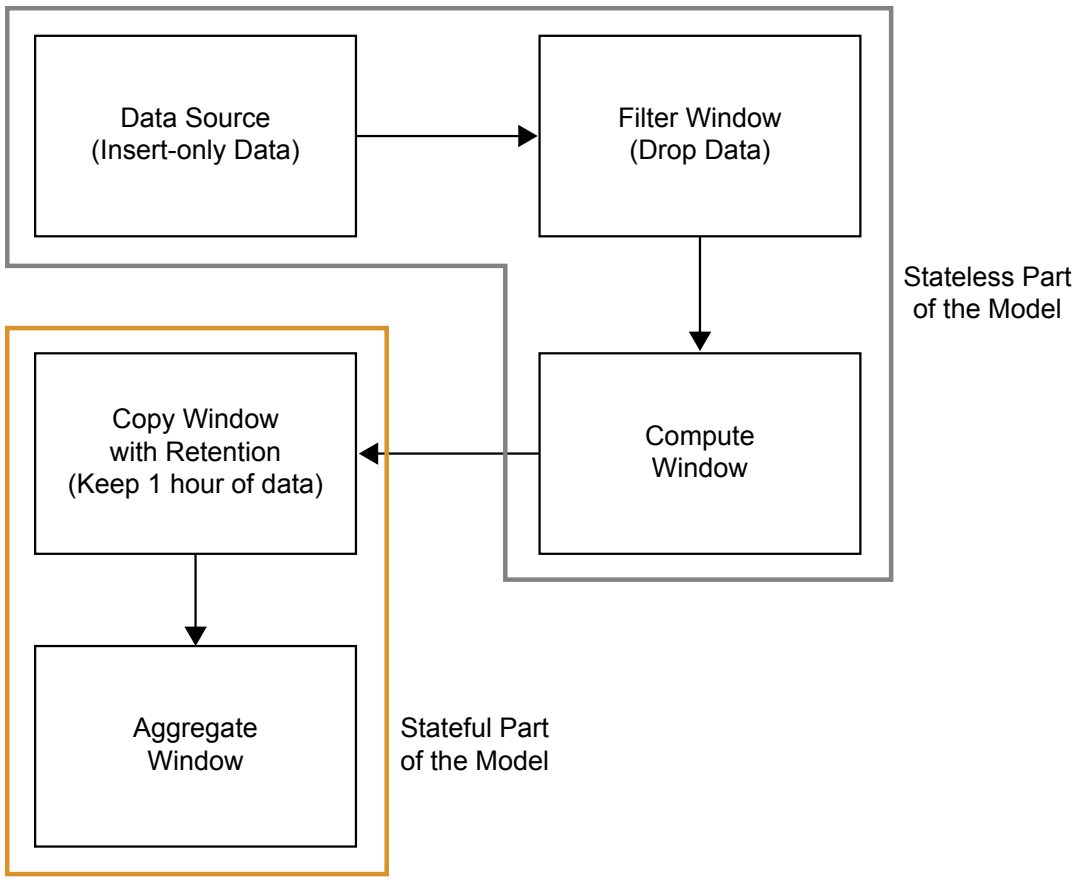
Here the data source is purely through Inserts. Therefore, the model can be made stateless by using an index type of **`pi_EMPTY`**. The filter receives inserts from the source, and drops some of them based on the filter criteria, so it produces a set of inserts as output. Thus, the filter can be made stateless also by using an index type of **`pi_EMPTY`**.

The compute window transforms the incoming inserts by selecting some of the output fields of the input events. The same window computes other fields based on values of the input event. It generates only inserts, so it can be stateless.

After the compute window, there is an aggregate window. This window type needs to retain events. Aggregate windows group data and compress groups into single events. If an aggregate window is fed a stream of Inserts, it would grow in an unbounded way.

To control this growth, you can connect the two sections of the model with a copy window with a retention policy.

Figure 22.2 *Modified Event Stream Processing Model with Stateless and Stateful Parts*



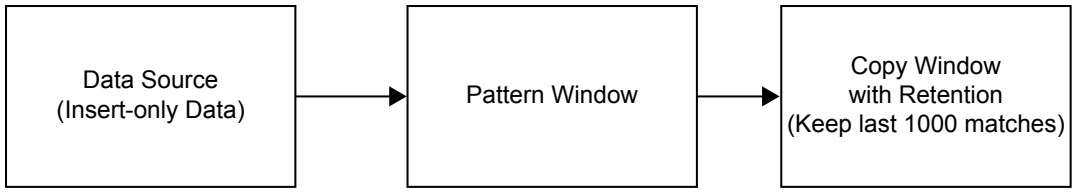
The stateful part of the model is accurately computed, based on the rolling window of input data. This model is bounded in memory.

Controlling Pattern Window Matches

Pattern matches that are generated by pattern windows are Inserts. Suppose you have a source window feeding a pattern window. Because a pattern window generate Inserts only, you should make it stateless by specifying an index type of **`pi_EMPTY`**. This prevents the pattern window from growing infinitely. Normally, you want to keep some of the more recent pattern matches around. Because you do not know how frequent the pattern generates matches, follow the pattern window with a count-based copy window.

Suppose you specify to retain the last 1000 pattern matches in the copy window.

Figure 22.3 *Event Stream Processing Model with Copy with Retention*



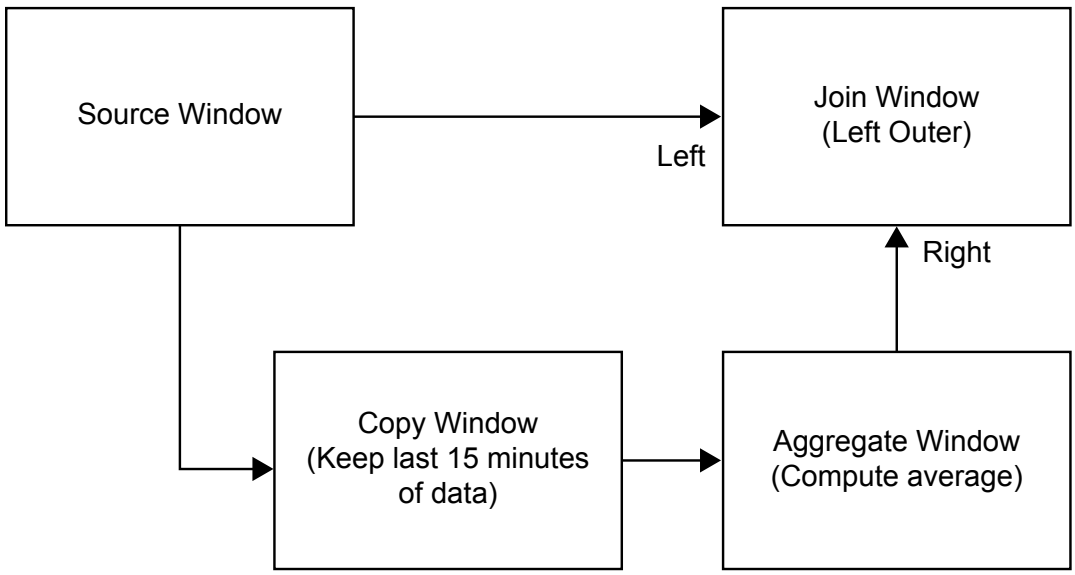
In cases like these, it is more likely that the copy window is queried from the outside using adapters, or publish/subcribe clients. The copy window might also feed other sections of the model.

Augmenting Incoming Events with Rolling Statistics

Suppose you have an insert stream of events, and one or more values are associated with the events. You want to augment each input event with some rolling statistics and then output the augmented events. Solving this problem requires using advanced features of the modeling environment.

For example, suppose you have a stream or stock trades coming in and you want to augment them with the average stock price in the past. You build the following model.

Figure 22.4 *Event Stream Processing Model Using Advanced Features*



To control the aggregate window:

- Put retention before it (the copy window).
- Group it by symbol (which is bounded), and use the additive aggregation function average (**`ESP_aAve`**), which does not need to store each event for its computation.

The join window can be problematic. Ordinarily you think of a join window as stateful. A join retains data for its fact window or dimension window and performs matches. In this case, you want a special, but frequently occurring behavior. When input comes in, pause it at the join until the aggregate corresponding to the input is processed. Then link the two together, and pass the augmented insert out.

To process input in this way:

1. Make the join a left outer join, with the source feed the left window, and the aggregate feeds the right window.
2. Set Tagged Token data flow model on for the projects. This turns on a special feature that causes a fork of a single event to wait for both sides of the fork to rejoin before generating an output event.
3. Set the index type of the join to **`pi_EMPTY`**, making the join stateless. A stateless left outer join does not create a local copy of the left driving window (FACT window). It does not keep any stored results of the join. However, there is always a reference-counted copy the lookup window. In the case of a left outer join, this is the right window. The lookup window is controlled by retention in this case, so it is bounded.
4. Ordinarily, a join, when the dimension window is changed, tries to find all matching events in the fact window and then issue updates for those joined matches. You do not want this behavior, because you are matching events in lock step. Further, it is simply not possible because you do not store the fact data. To prevent this regeneration on each dimension window change, set the **`no-regenerates`** option on the join window.

In this way you create a fast, lightweight join. This join stores only the lookup side, and produces a stream of inserts on each inserted fact event. A stateless join is possible for left and right outer joins.

The following XML code implements this model.

```
<engine port='52525' dateformat='%d/%b/%Y:%H:%M:%S'>
    <projects>
        <project name='trades_proj' pubsub='auto'
                 use-tagged-token='true' threads='4'>
            <contqueries>
                <contquery name='trades_cq'>

                    <windows>
                        <window-source name='Trades'
                                       index='pi_RBTREE'>
                            <schema>
                                <fields>
                                    <field name='tradeID' type='string' key='true'/>
                                    <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-source>

                        <window-copy name='TotalIn'>
                            <retention type='bycount_sliding'>5</retention>
                        </window-copy>
```

```
                    <window-aggregate name='RunTotal'>
                        <schema>
                            <fields>
                                <field name='tradeID' type='string'/>
                                <field name='security' type='string' key='true'/>
                                <field name='quantityTotal' type='double'/>
                            </fields>
                        </schema>
                        <output>
                            <field-expr>ESP_aLast(tradeID)</field-expr>
                            <field-expr>ESP_aSum(quantity)</field-expr>
                        </output>
                    </window-aggregate>

                    <window-join name='JoinTotal'
                                 index='pi_EMPTY'>
                        <join type="leftouter"
                              no-regenerates='true'>
                            <conditions>
                              <fields left='tradeID' right='tradeID'/>
                              <fields left='security' right='security'/>
                            </conditions>
                        </join>
                        <output>
                            <field-selection name='quantity'
                              source='l_quantity'/>
                            <field-selection name='price'
                              source='l_price'/>
                            <field-selection name='traderID'
                              source='l_traderID'/>
                            <field-selection name='time'
                              source='l_time'/>
                            <field-selection name='quantityTotal'
                              source='r_quantityTotal'/>
                        </output>
                    </window-join>
                </windows>

                <edges>
                    <edge source='Trades'
                          target='JoinTotal'/>
                    <edge source='Trades'
                          target='TotalIn'/>
                    <edge source='TotalIn'
                          target='RunTotal'/>
                    <edge source='RunTotal'
                          target='JoinTotal'/>
                </edges>

            </contquery>
        </contqueries>
    </project>
</projects>
</engine>
```

Appendix 1

Supported Functions

Dictionary **367**
ABS 367
AND 367
BASE64DECODE 368
BASE64ENCODE 368
BETWEEN 369
BOOLEAN 369
CEILING 370
COMPARE 371
CONCAT 371
CONCATDELIM 372
CONTAINS 372
DECREMENT 373
DIFF 373
EQUALS 374
EVENTNUMBER 374
FALSE 375
FLOOR 375
GT 376
GTE 376
GUID 377
INCREMENT 377
IF 378
IFNEXT 378
INDEX 379
INDEXOF 380
INPUT 380
INTEGER 380
ISNULL 381
ISSET 381
INTERVAL 382
JSON 383
LASTINDEXOF 383
LISTITEM 384
LISTSIZE 384
LT 385
LTE 385
MAPVALUE 386
MAPVALUES 387
MAX 387
MEAN 388

MIN . . . 388
MOD . . . 389
NEG . . . 389
NORMALIZESPACE . . . 390
NEQUALS . . . 390
NOT . . . 391
NUMBER . . . 391
OR . . . 392
OUTPUT . . . 392
OUTSTR . . . 393
PRECISION . . . 393
PRODUCT . . . 394
QUOTIENT . . . 394
RANDOM . . . 394
RGX . . . 395
RGXINDEX . . . 395
RGXLASTTOKEN . . . 396
RGXMATCH . . . 397
RGXREPLACE . . . 397
RGXREPLACEALL . . . 398
RGXTOKEN . . . 398
RGXV . . . 399
ROUND . . . 399
SETCONTAINS . . . 400
STARTSWITH . . . 400
STRING . . . 401
STRINGLENGTH . . . 401
STRIP . . . 402
SUBSTRING . . . 402
SUBSTRINGAFTER . . . 403
SUBSTRINGBEFORE . . . 403
SUM . . . 404
SWITCH . . . 404
SYSTEMMICRO . . . 405
SYSTEMMILLI . . . 405
TIMECURRENT . . . 406
TIMEDAYOFMONTH . . . 406
TIMEDAYOFWEEK . . . 406
TIMEDAYOFYEAR . . . 407
TIMEGMTTOLOCAL . . . 407
TIMEGMTSTRING . . . 407
TIMEHOUR . . . 408
TIMEMINUTE . . . 408
TIMEMINUTEOFDAY . . . 409
TIMEPARSE . . . 409
TIMESECOND . . . 410
TIMESTAMP . . . 410
TIMESTRING . . . 410
TIMESECONDOFDAY . . . 411
TIMETODAY . . . 411
TIMEYEAR . . . 412
TOLOWER . . . 412
TOUPPER . . . 412
TRANSLATE . . . 413
TRUE . . . 413
URLDECODE . . . 414

URLENCODE 414
XPATH 415

Dictionary

ABS

Returns the absolute floating-point value of the supplied argument.

Syntax

abs(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
abs(-55)=55
abs(44)=44
```

AND

When both arguments are true, returns true. Otherwise, returns false.

Syntax

and(*argument1*, *argument2*)

Required Argument

argument1*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.

- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.
- If the argument returns a numeric value, then the function returns true when the numeric value is nonzero, otherwise it returns false.
- If the argument returns a string value:
  - When the string value is 'true', the function returns true.
  - When the string value is 'false', the function returns false.
  - Else, when the length of the string is > 0, the function returns true, otherwise false.

Example

```
and(gt(3,2),gt(2,5))=0
and(gt(3,2),gt(12,5))=1
and(0,1)=0
and('non empty string',55)=1
and('true',55)=1
and('',4)=0
```

BASE64DECODE

Decodes the supplied base64-encoded string.

Syntax

base64Decode(*string*)

Required Argument

string
specifies a base64–encoded string. There is no length limit to this string.

Example

```
base64Decode('dGhpcyBpcyBhIHRlc3Q=')=this is a test
```

BASE64ENCODE

Encodes the supplied string into a base64–encoded string.

Syntax

base64Encode(*string*)

Required Argument

string
specifies a text string. There is no length limit to this string.

Example

```
base64Encode('this is a test')=dGhpcyBpcyBhIHRlc3Q=
```

BETWEEN

If the first argument is greater than the second and less than the third, returns true. Otherwise, returns false.

Syntax

between(*argument1*, *argument2*, *argument3*)

Required Argument

argument1*, *argument2*, *argument3
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
between(20,17,30)=1
between(20,17,15)=0
```

BOOLEAN

If the supplied argument is a string, returns true when the string has length greater than 0. If the supplied argument is numeric, returns true when value is not equal to 0. If the supplied argument is a Boolean expression, returns true when the value is true. Otherwise, returns false.

Syntax

boolean(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Note: The special string values **'true'** and **'false'** are handled outside the string length > 0. If **'true'**, the function returns 1. If **'false'**, the function returns 0.

Example

```
boolean('my string')=1
boolean('')=0
boolean(10)=1
boolean(0)=0
boolean(gt(4,7))=0
boolean(gt(7,5))=1
```

CEILING

Returns the integer value above the numeric value of the supplied argument.

Syntax

ceiling(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
ceiling(product(4,4.1))=17
```

COMPARE

Compares the first argument to the second. If the first argument is less than the second, then it returns -1. If the first is greater than the second, then it returns 1. If the first is equal to the second, then it returns 0.

Syntax

compare(*argument1*, *argument2*)

Required Argument

argument1*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

The type of the first argument determines whether equality is determined by a string or numeric comparison.

Example

```
compare('bears','lions')=-1
compare('lions','bears')=1
compare('bears','bears')=0
compare(10,20)=-1
compare(20,10)=1
compare(10,10)=0
```

CONCAT

Returns a string that is the concatenation of the string values of the supplied arguments.

Syntax

concat(*argument1*, *argument2*,…<*argumentN*>)

Required Argument

argument1*, *…argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.

- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

A minimum of two arguments is required.

Example

```
concat('Name: ','Joe',', Age: ',floor(sum(25,10)),'.') = Name: Joe, Age: 35.
```

CONCATDELIM

Returns a string that is the concatenation of the supplied values separated by the specified delimiter.

Syntax

concatDelim(‘*delimiter*’,*argument1*, *argument2*, …<*argumentN*>)

Required Arguments

‘*delimiter*’
specifies a character used as a delimiter.

argument1*, *…argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
concatDelim('.','www','sas','com') = www.sas.com
```

CONTAINS

If the string value of the first argument contains the string value of the second, then it returns true. Otherwise, it returns false.

Syntax

contains(*argument1*, *argument2*)

Required Argument

argument1*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
contains('www.sas.com','sas') = true
contains('www.google.com','sas') = false
```

DECREMENT

Returns the numeric value of the supplied argument minus 1. This function supports only integers.

Syntax

decrement(*argument*)

Required Argument

argument
specifies an integer.

Example

```
decrement(10)=9
```

DIFF

Returns the value of the first argument minus the second.

Syntax

diff(*argument1*, *argument2*)

Required Argument

argument1*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.

- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
diff(sum(8,7),22)=-7.0
```

EQUALS

Returns true if the first argument is equal to the second. Otherwise, it returns false.

Syntax

equals(*argument1*, *argument2*)

Required Argument

argument1*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

The type of the comparison depends on the type of the first argument.

Example

```
equals('sas.com',string('sas','.com'))=1
equals('sas.com','google.com')=0
equals(10,sum(5,5))=1
```

EVENTNUMBER

Returns the 0–based number of events that are generated by the current incoming event. The number is incremented each time that the function is invoked.

Syntax

eventNumber()

Example

```
string($id,'-',eventNumber())=eventid-0
string($id,'-',eventNumber())=eventid-1
string($id,'-',eventNumber())=eventid-2
```

FALSE

Returns true if the Boolean value of the argument is false. Otherwise, it returns true.

Syntax

false(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
false(0)=1
false(equals(10,10))=0
```

FLOOR

Returns the integer value below the numeric value of the supplied argument.

Syntax

floor(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.

- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
floor(product(3.5,7))=24
```

GT

Returns true if the first argument is greater than the second. Otherwise, it returns false.

Syntax

gt(*argument*, *argument2*)

Required Argument

argument*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

The type of the comparison depends on the type of the first argument.

Example

```
gt(sum(10,4),13)=true
gt('internet explorer','internet explorer')=false
gt('internet explorer','netscape')=false
```

GTE

Returns true if the first argument is greater than or equal to the second. Otherwise, it returns false.

Syntax

gt(*argument*, *argument2*)

Required Argument

argument*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

The type of the comparison depends on the type of the first argument.

Example

```
gte(sum(10,4),13)=true
gte('internet explorer','internet explorer')=true
gte('netscape','internet explorer')=true
```

GUID

Returns a globally unique identifier.

Syntax

guid()

Example

```
guid()=46ca7b9e-b11d-41be-a3eb-be8bc8553aed
guid()=319cd2a6-1b30-4c1b-8ac7-55e9465ea066
```

INCREMENT

Returns the numeric value of the first argument + 1. This function only supports integers.

Syntax

increment(*argument*)

Required Argument

argument
specifies an integer value or a function that returns an integer value.

Example

```
increment(10)=11
```

IF

If the Boolean value of the first argument is true, returns the second argument. Otherwise, it returns the third argument if specified.

Syntax

if(*argument1* , *argument2*, <*argument3*>)

Required Arguments

argument1
specifies a Boolean expression

***argument2*,**
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Optional Argument

argument3
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
if(equals('x','x'),'one','two')=one
if(equals('x','y'),'one','two')=two
if(equals('x','y'),'one')=
```

IFNEXT

Evaluates the first argument in a pair. When the argument evaluates to true, the function returns the value of the second argument in the pair.

Syntax

ifNext(*argument* , *argument2*, …<*argumentN*>, <*argumentN+1*>)

Required Arguments

argument
specifies a Boolean expression.

argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
ifNext(gt(20,10),'value 1',lt(20,10),'value 2')=value 1
ifNext(gt(20,100),'value 1',lt(20,100),'value 2')=value 2
```

INDEX

Returns the value of argumentN, where N is the numeric value of specified index.

Syntax

index(*index* , *argument0*, …<*argumentN*>)

Required Arguments

index
specifies an integer or a function that returns an integer.

argument0*, *argument1*, …*argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

The minimum number of arguments is 2.

Example

```
index(1,'larry','moe','curly')=moe
index(random(0,4),10,20,30,40,50)=40
index(random(0,4),10,20,30,40,50)=20
```

INDEXOF

Returns the 0–based index of the string value of the first argument in the string value of the second argument. Returns -1 if the value is not found.

Syntax

indexOf(*argument1* , *argument2*)

Required Argument

argument1*, *argument2
specifies a string.

Example

```
indexOf('SAS Event Stream Processing','Stream')=10
indexOf('SAS Event Stream Processing','Google')=-1
```

INPUT

Returns the name of the event stream processing input window.

Syntax

input()

Example

```
input()=sourceWindow
```

INTEGER

Returns the integer value of the argument.

Syntax

integer(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
integer('88.45')=88
integer(111.23)=111
```

ISNULL

Returns true if the argument is not set, otherwise it returns false.

Syntax

isNull(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
isNull($unresolved)=1
isNull('my string')=0
```

ISSET

Returns true if the argument is set, otherwise it returns false.

Syntax

isSet(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
isSet($unresolved)=0
isSet('my string')=1
```

INTERVAL

Takes the numeric value of the first argument and compares it to the numeric values of all remaining arguments. If the numeric value of the first argument is less than one of the arguments that follow, then the value of the argument that follows that one is returned.

Syntax

interval(*argument* , *argument2*, *argument3*, …<*argumentN*>)

Required Argument

arguments
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
interval(85,60,'F',70,'D',80,'C',90,'B','A')=B
interval(90,60,'F',70,'D',80,'C',90,'B','A')=A
```

JSON

Parses the JSON object specified in the first argument and returns a value as a function of the second argument.

Syntax

json(*argument* , *argument2*)

Required Arguments

argument
specifies a JSON object.

argument2
specifies an evaluation string. In a name value pair, specify the name of the object whose value you want to return.

Example

```
json('{first:"john",last:"smith",hobbies:["running","reading","golf"]}',
             'first')=john
json('{first:"john",last:"smith",hobbies:["running","reading","golf"]}',
     'last')=smith
json('{first:"john",last:"smith",hobbies:["running","reading","golf"]}',
     'hobbies[1]')=reading
json(#myJson,'hobbies[1]')=reading
```

LASTINDEXOF

Returns the last index of the string value of the second argument in the string value of the first, or -1 if the value is not found.

Syntax

lastIndexOf(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
lastIndexOf('http://www.sas.com/products/webanalytics','/')=27
```

LISTITEM

This function has two uses. 1) Parses the first argument using the specified delimiter and then returns the value at the specified index. 2) References an existing list in the specified function context and returns its value at the specified index.

Syntax

listItem(*argument* , *delimiter*, *index*)

listItem(*reference* , *index*)

Required Arguments

argument
specifies a delimited string.

delimiter
specifies a character used as a delimiter.

index
specifies a numeric value used as an index.

reference
specifies a reference to a function context.

Example

```
listItem('one,two,three,four',',',2)=three
listItem(#myList,0)=one
```

LISTSIZE

This function has two uses. 1) Parses the first argument using the specified delimiter and then returns its size. 2) References an existing list in the specified function context and returns its size.

Syntax

listSize(*argument* , *delimiter*)

listSize(*reference*)

Required Arguments

argument
specifies a delimited string.

delimiter
specifies a character used as a delimiter.

reference
specifies a reference to a function context.

Example

```
listSize('one,two,three,four',',')=4
listSize(#myList)=4
```

LT

Returns true if the first argument is less than the second. Otherwise, returns false.

Syntax

lt(*argument* , *argument2*)

Required Arguments

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
lt(sum(10,4),13)=false
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

LTE

Returns true if the first argument is less than or equal to the second. Otherwise, returns false.

Syntax

lt(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
lte(sum(10,3),13)=true
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

MAPVALUE

This function has two uses. 1) Parses name-value pairs from the first argument using the specified outer delimiter and the specified inner delimiter, and then extracts the value for the specified name. 2) References an existing value map in the referenced function context, and then extracts the value for the name.

Syntax

mapValues(*argument* , *outerdelimiter*, *innerdelimiter*, *delimiter*, *name*)
mapValues(*#reference*, *name*)

Required Arguments

argument
specifies a delimited string of name-value pairs.

outerdelimiter*, *innerdelimiterdelimiter
specifies characters used as delimiters.

name
specifies the name in the name-value pair specified in *argument*.

#reference
specifies a reference to a function context.

Example

```
mapValue('first:John;last:Doe;occupation:plumber',';',':','occupation')=plumber
mapValue(#myMap,'occupation')=plumber
```

MAPVALUES

This function has two uses. 1) Parses name-value pairs from the first argument using the specified outer delimiter and the specified inner delimiter, and then extracts the values for each specified name. 2) References an existing value map in the referenced function context and extracts the values for each specified name.

Syntax

mapValues(*argument* , *outerdelimiter*, *innerdelimiter*, *delimiter*, *name1*,…<*nameN*>)

mapValues(*#reference*, *name1*, …<*nameN*>)

Required Arguments

argument
specifies a delimited string of name-value pairs.

outerdelimiter*, *innerdelimiterdelimiter
specifies characters used as delimiters.

name1*, *…nameN
specifies the name in the name-value pair specified in *argument*.

#reference
specifies a reference to a function context.

Example

```
mapValues('first=John,last=Doe',',','=',':','first','last')=John:Doe
mapValues(#myMap,'first','last')=John:Doe
```

MAX

Returns the largest numeric value of all specified arguments.

Syntax

max(*argument* , *argument2*, …<*argumentN*>)

Required Argument

argument*, *argument2*, *…argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.

- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

The minimum number of arguments is 1.

Example

```
max(33,44.2,sum(1,3,2,12),-33.21)=44.2
```

MEAN

Returns the mean value of all specified arguments.

Syntax

mean(*argument* , *argument2*, …<*argumentN*>)

Required Argument

argument*, *argument2*, …*argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
mean(33,44.2,sum(1,3,2,12),-33.21)=15.4975
```

MIN

Returns the smallest numeric value of all specified arguments.

Syntax

min(*argument* , *argument2*, …<*argumentN*>)

Required Argument

argument*, *argument2*, …*argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

The minimum number of arguments is 1.

Example

```
min(33,44.2,sum(1,3,2,12),-33.21)=-33.21
```

MOD

Returns the remainder of the first argument divided by the second.

Syntax

mod(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
mod(10,3)=1.0
```

NEG

Returns the negative numeric value of the specified argument.

Syntax

neg(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
neg(55)=-55
```

NORMALIZESPACE

Returns a string that is created by replacing any extra white space in the specified argument with a single space.

Syntax

normalizeSpace(*argument*)

Required Argument

argument
specifies a string.

Example

```
normalizeSpace('Sentence   with   many  spaces')=Sentence with many spaces
```

NEQUALS

Returns true if the first argument is not equal to the second. Otherwise, returns false.

Syntax

nequals(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.

- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
nequals('sas.com',string('sas','.com'))=0
nequals('sas.com','google.com')=1
nequals(10,sum(5,5))=0
```

NOT

Returns true if the Boolean value of the argument is false. Otherwise, returns false.

Syntax

not(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
not(0)=1
not(equals(10,10))=0
```

NUMBER

Returns the numeric value of the argument.

Syntax

number(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
number('88.45')=88.45
number(111.23)=111.23
number(gt(2,1))=1
```

OR

Returns true if any of the supplied arguments are true. Otherwise, returns false.

Syntax

or(*argument* , *argument2*, …<*argumentN*>)

Required Argument

argument*, *argument2*, …*argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

The minimum number of arguments is 1.

Example

```
or(equals('a','b'),nequals('a','b'))=1
or(equals('a','b'),nequals('a','a'))=0
```

OUTPUT

Returns the name of the event stream processing output window.

Syntax

output()

Example

```
input()=myFunctionalWindow
```

OUTSTR

If the argument contains a string or one of a group of strings, returns the associated value. If there are no matches, returns a specified default value.

Syntax

outstr(*argument* , *string1*, *value_associated_with_string1*, …<*stringN*>, <*value_associated_with_stringN*>, *default*)

Required Arguments

argument
specifies a string.

string1…stringN
specifies a string or group of strings.

value_associated_with_string1…value_associated_with_stringN
specifies a string or group of strings.

default
specifies a string or group of strings.

Example

```
outstr('government spending','govern','Government','Other')=Government
outstr('spending',('govern','spend'),
       'Government or Spending','Other')=Government or Spending
outstr('bob',('govern','spend'),'Government or Spending',
      ('john','jack','bob'),'Names','Other')=Names
outstr('stream processing',('govern','spend'),'Government or Spending',
       ('john','jack','bob'),'Names','Other')=Other
```

PRECISION

Sets the decimal point precision of the first argument to the second argument.

Syntax

precision(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies a numeric value.

Example

```
precision(123.44567,2)=123.45
```

PRODUCT

Returns the product of the supplied arguments.

Syntax

product(*argument* , *argument2*…<*argumentN*>)

Required Argument

argument*, *argument2*, … *argumentN
specifies a numeric value or a function that returns a numeric value.

Example

```
product(3,sum(2,4),2)=36
```

QUOTIENT

Returns the quotient of the supplied arguments.

Syntax

quotient(*argument* , *argument2*…<*argumentN*>)

Required Argument

argument*, *argument2*, … *argumentN
specifies a numeric value or a function that returns a numeric value.

Example

```
quotient(3,sum(2,4),2)=0.25
```

RANDOM

Returns a random number between the first argument and the second.

Syntax

random(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies a numeric value.

Example

```
random(100,1000)=741
random(100,1000)=356
random(100,1000)=452
precision(random(0,.5),2)=0.17
```

RGX

Runs the specified regular expression on a supplied string and returns the result. If a group is specified, the result is the content of the specified numeric regular expression group.

Syntax

rgx(*regular_expression* , *string*…<*group*>)

Required Arguments

regular_expression
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

Optional Argument

group
specifies a numeric reference to the regular expression.

Example

```
rgx('.*/view/([0-9]*)/([0-9]*)',
           'http://cistore-dev.unx.sas.com/products/view/23/4',
           1)=23
rgx(#myExpr,
    'http://cistore-dev.unx.sas.com/products/view/23/4'
    ,2)=4
```

RGXINDEX

Runs the specified regular expression on a supplied string. When a match is found, returns the index of the match. If no match is found, returns -1.

Syntax

rgxIndex(*regular_expression* , *string…<stringN>*)

Required Arguments

regular_expression
specifies a regular expression or a reference to a regular expression in the function context.

string…stringN
specifies a string.

Example

```
rgxIndex('developer','larry - manager','moe - tester','curly - developer')=2
```

RGXLASTTOKEN

Uses the regular expression in the first argument as a delimiter within the regular expression of the second to find all strings separated by that expression.

Syntax

rgxLastToken(*regular_expression1* , *regular_expression2…<index_value>*)

Required Arguments

regular_expression1
specifies a regular expression or a reference to a regular expression in the function context.

regular_expression2
specifies a regular expression.

Optional Argument

index_value
specifies an index value (defaults to 0) that counts from the last token in the expression. When this value is greater than 0 and less than or equal to the number of tokens in the regular expression, the token at the value is returned. Otherwise, null is returned.

Example

```
rgxLastToken('/','data/opt/sas/dataflux')=dataflux
rgxLastToken('/','data/opt/sas/dataflux',2)=opt
rgxLastToken('\.','www.sas.com')=com
```

RGXMATCH

Compares the regular expression in the first argument to the second argument and returns a Boolean value that indicates whether a match is found.

Syntax

rgxMatch(*regular_expression1* , *string…<group>*)

Required Arguments

regular_expression1
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

Optional Argument

group
specifies a numeric reference to the regular expression. When specified, the result is the content of the specified numeric regular expression group.

Example

```
rgxMatch('(google|yahoo|bing)','http://www.google.com')=1
rgxMatch('(google|yahoo|bing)','http://www.sas.com')=0
```

RGXREPLACE

Parses the regular expression in the first argument against the string in the second argument and replaces the first match with the string in the third argument.

Syntax

rgxReplace(*regular_expression1* , *string1*, *string2*)

Required Arguments

regular_expression1
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

string2
specifies a string.

Example

```
rgxReplace('(google|yahoo|bing)','http://www.google.com','sas')=http://www.sas.com
```

RGXREPLACEALL

Parses the regular expression in the first argument against the string in the second argument and replaces any match with the string in the third argument.

Syntax

rgxReplaceAll(*regular_expression1* , *string1*, *string2*)

Required Arguments

regular_expression1
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

string2
specifies a string.

Example

```
rgxReplaceAll('(google|yahoo|bing)','http://www.google.com/google/products','sas')
       =http://www.sas.com/sas/products
```

RGXTOKEN

Uses the first argument as a delimiter within the second argument to find all strings separated by that delimiter.

Syntax

rgxToken(*delimiter* , *string*, <*index*>)

Required Arguments

delimiter
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

index
specifies a numeric value that serves as an index when parsing the string. If the index is less than or equal to the number of tokens in the string, the token at the index value is returned. Otherwise, null is returned. The default value is 0.

Example

```
rgxToken('/','data/opt/sas/dataflux')=data
rgxToken('/','data/opt/sas/dataflux',2)=sas
rgxToken('\.','www.sas.com')=www
```

RGXV

Parses the regular expression in the first argument against the string in the second argument and returns all matches delimited by the specified delimiter.

Syntax

rgxV(*regular_expression* , *string*, *delimiter*<*group*>)

Required Arguments

regular_expression
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

delimiter
specifies a character value that serves as a delimiter when parsing *string*.

Optional Argument

group
specifies a numeric reference to the regular expression. When specified, the result is the content of the specified numeric regular expression group.

Example

```
rgxV('(jerry|scott|vince)',
            'The ESP product has jerry, scott, and vince working on it',
            ' : ')=jerry : scott : vince
```

ROUND

Returns the rounded numeric value of the argument.

Syntax

round(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **xpath** function to refer to an XML object named myXML, you would specify this: **#myXML**.
- a function.

Example

```
round(34.56)=35
round(34.46)=34
```

SETCONTAINS

This function has two uses. 1) Parses a specified set of tokens containing the specified delimiter to check whether a specified string appears within the set. 2) References an existing set of tokens in the function context to check whether a specified string appears in the set.

Syntax

setContains(*set_of_tokens* , *delimiter*, *string*)

setContains(*#reference string*)

Required Arguments

set_of_tokens
specifies a string of tokens.

delimiter
specifies a character value used as a delimiter.

string
specifies a string.

reference
specifies a reference to a function context.

Example

```
setContains('one,two,three,four',',','two')=1
setContains(#mySet,'five')=0
```

STARTSWITH

Returns true if the first argument starts with the second. Otherwise, returns false.

Syntax

startsWith(*argument1* , *argument2*)

Required Argument

argument1*, *argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
startsWith('www.sas.com','www.')=1
startsWith('www.sas.com','sww.')=0
```

STRING

Returns the string value of the argument.

Syntax

string(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
string(33.9)=33.9
```

STRINGLENGTH

Returns the length of the string value of the argument.

Syntax

stringLength(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
stringLength('SAS ESP XML')=11
```

STRIP

Returns the string created after removing leading or trailing white space from the argument.

Syntax

strip(*argument*)

Required Argument

argument
specifies a string.

Example

```
strip('          SAS ESP XML     ')=SAS ESP XML
```

SUBSTRING

Returns the string created by taking the substring of the value of the first argument at the specified index.

Syntax

substring(*argument* , *index*, <*length*>)

Required Arguments

argument
specifies a string.

index
specifies a numeric value that defines an index with which to parse the *argument*.

Optional Argument

length
specifies a numeric value. If specified, the substring is *length* size, otherwise it contains all characters to the end of the string.

Example

```
substring('www.sas.com',4,3)=sas
substring('www.sas.com',4)=sas.com
```

SUBSTRINGAFTER

Returns the string that results from taking the value of the first argument after an occurrence of the value of the second argument.

Syntax

substringAfter(*argument1* , *argument2*, <*index*>)

Required Argument

argument1*, *argument2
specifies a string.

Optional Argument

index
specifies a numeric value that defines an index with which to parse *argument1*. When specified, the content after that occurrence is returned.

Example

```
substringAfter('www.sas.com','.')=sas.com
substringAfter('www.sas.com','.',2)=com
```

SUBSTRINGBEFORE

Returns the string that results from taking the value of the first argument before an occurrence of the value of the second argument.

Syntax

substringBefore(*argument1* , *argument2*, <*index*>)

Required Argument

argument1*, *argument2
specifies a string.

Optional Argument

index
specifies a numeric value that defines an index with which to parse *argument1*. When specified, the content after that occurrence is returned.

Example

```
substringBefore('www.sas.com','.')=www
substringBefore('www.sas.com','.',2)=www.sas
```

SUM

Returns the sum of the numeric values of all arguments.

Syntax

sum(*argument* , *argument2*…<*argumentN*>)

Required Argument

argument*, *argument2*, … *argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
sum(33,22,55.4,34,min(0,4,-9))=135.4
```

SWITCH

Parses arguments beginning with the second one. When an argument matches the first, it returns the following argument. If no match is found, it returns null.

Syntax

switch(*argument* , *argument2*, …<*argumentN*>, <*argumentN+1*>)

Required Argument

argument*, *argument2*…*argumentN+1
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
switch('bob','jerry','manager','bob','developer')=developer
switch('jerry','jerry','manager','bob','developer')=manager
switch('moe','jerry','manager','bob','developer')=
```

SYSTEMMICRO

Returns the number of microseconds since Jan 1, 1970.

Syntax

systemMicro()

Example

```
systemMicro()=1420557039483912
```

SYSTEMMILLI

Returns the number of milliseconds since Jan 1, 1970.

Syntax

systemMilli()

Example

```
systemMilli()=1420557039483
```

TIMECURRENT

Returns the current time.

Syntax

timeCurrent()

Example

```
timeCurrent()=1421157236
timeString(timeCurrent())=Tue Jan 13 08:53:56 2015
```

TIMEDAYOFMONTH

Returns the day of the month of the current or specified time.

Syntax

timeDayOfMonth(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeDayOfMonth()=13
timeDayOfMonth(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=21
```

TIMEDAYOFWEEK

Returns the day of the week of the current or specified time.

Syntax

timeDayOfWeek(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeDayOfWeek()=2
timeDayOfWeek(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=0
```

TIMEDAYOFYEAR

Returns the day of the year of the current or specified time.

Syntax

timeDayOfYear(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeDayOfYear()=12
timeDayOfYear(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=171
```

TIMEGMTTOLOCAL

Converts the GMT that is specified in the argument to local time.

Syntax

timeGmtToLocal(*argument*)

Required Argument

argument
specifies a time value or a function that returns a time value.

Example

```
timeString(timeGmtToLocal(timeCurrent()))=Tue Jan 13 02:10:08 2015
```

TIMEGMTSTRING

Outputs the GMT time represented by the first argument.

Syntax

timeGmtString(*argument*, <*argument2*>)

Required Argument

argument
specifies a time value or a function that returns a time value.

Optional Argument

argument2
specifies a time format.

Example

```
timeString(timeCurrent(),'%Y-%m-%d %H:%M:%S %Z')=2015-02-20 07:57:18 EST
timeGmtString(timeCurrent(),'%Y-%m-%d %H:%M:%S %Z')=2015-02-20 12:57:18 GMT
```

TIMEHOUR

Returns the hour of the day of the current or specified time.

Syntax

timeHour(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeHour()=9
timeHour(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=14
```

TIMEMINUTE

Returns the minute of the hour of the current or specified time.

Syntax

timeMinute(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeMinute()=22
timeMinute(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=45
```

TIMEMINUTEOFDAY

Returns the minute of the day of the current or specified time.

Syntax

timeMinuteOfDay(*<argument>*)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeMinuteOfDay()=563
timeMinuteOfDay(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=885
```

TIMEPARSE

Returns a string that represents the time specified in the first argument.

Syntax

timeParse(*time*,*<format>*)

Required Argument

time
specifies a time specification or a function that returns a time specification.

Optional Argument

format
specifies a time format that is supported by the UNIX **`strftime`** function.

Example

```
timeParse(timeString())=1421159135
timeParse('01/01/2015 00:00:00','%m/%d/%Y %H:%M:%S')=1420088400
```

TIMESECOND

Returns the second of the minute of the current or specified time.

Syntax

timeSecond(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeSecond()=37
timeSecond(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=15
```

TIMESTAMP

Returns the current time as a string.

Syntax

timeStamp(<*format*>)

Optional Argument

format
specifies a time format that is supported by the UNIX **`strftime`** function.

Example

```
timeStamp()=Thu Feb 19 15:09:35 2015
timeStamp('%m-%d-%Y')=02-19-2015
```

TIMESTRING

Returns the time represented by the first argument.

Syntax

timeString(*time*,<*format*>)

Required Argument

time
specifies a time specification or a function that returns a time specification.

Optional Argument

format
specifies a time format. If you do not specify *format*, the system default time format is used.

Example

```
timeStamp()=Thu Feb 19 15:09:35 2015
timeStamp('%m-%d-%Y')=02-19-2015
```

TIMESECONDOFDAY

Returns the second of the day of the current or specified time.

Syntax

timeSecondofDay(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeSecondOfDay()=34035
timeSecondOfDay(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=53115
```

TIMETODAY

Returns a value that represents the first second of the current day relative to local time.

Syntax

timeToday()

Example

```
timeToday()=1421125200
```

TIMEYEAR

Returns the number of years since 1900 of the current or specified time.

Syntax

timeYear(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeYear()=115
timeYear(timeParse('06/21/2013 13:45:15','%m/%d/%Y %H:%M:%S'))=113
```

TOLOWER

Converts the value of the argument to lowercase.

Syntax

toLower(*string*)

Required Argument

string
specifies a string.

Example

```
toLower('Http://Www.Sas.Com/Products/Esp')=http://www.sas.com/products/esp
```

TOUPPER

Converts the value of the argument to uppercase.

Syntax

toUpper(*string*)

Required Argument

string
specifies a string.

Example

```
toUpper('Http://Www.Sas.Com/Products/Esp')=HTTP://WWW.SAS.COM/PRODUCTS/ESP
```

TRANSLATE

For each character in the second argument, finds the corresponding characters in the first argument and replaces them with the corresponding characters in the third.

Syntax

translate(*argument1*, *argument1*, *argument3*)

Required Argument

argument1*, *argument2*, *argument3
specifies a string. The length of *argument2* and *argument3* must be identical.

Example

```
translate('replace all vowels with its capital equivalent','aeiou','AEIOU')
          =rEplAcE All vOwEls wIth Its cApItAl EqUIvAlEnt
```

TRUE

Returns true if the Boolean value of the argument is true. Otherwise, it returns false.

Syntax

true(*argument*)

Required Argument

argument
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
true('testing')=1
true('')=0
true(gt(10,5))=1
```

URLDECODE

Decodes the URL represented by the argument.

Syntax

urlDecode(*argument*)

Required Argument

argument
specifies a string.

Details

For more information to encoding and decoding URLs, see this reference.

Example

```
urlDecode('http%3A%2F%2Fwww%2Esas%2Ecom%2Fproducts%2Fevent%20stream%20processing')
=http://www.sas.com/products/event stream processing
```

URLENCODE

Encodes the URL represented by the argument.

Syntax

urlEncode(*argument*)

Required Argument

argument
specifies a string.

Details

For more information to encoding and decoding URLs, see this reference.

Example

```
urlEncode('http://www.sas.com/products/event stream processing')
=http%3a%2f%2fwww%2esas%2ecom%2fproducts%2fevent%20stream%20processing
```

XPATH

Parses the XML in the first argument, evaluating the second argument in the XML context.

Syntax

xpath(*argument1*, *argument2*, <*argument3*>)

Required Arguments

argument1
specifies an instance of XML, represented by valid XML textual context or by a reference to XML elsewhere.

argument2
specifies an evaluation string.

Optional Argument

argument3
specifies a separator used when the function returns multiple results.

Example

```
xpath('<info><name>john smith</name><hobby>running</hobby>
              <hobby>reading</hobby><hobby>golf</hobby></info>',
             './/name/text()')=john smith
xpath(#myXml,'.//hobby/text()',')=running,reading,golf
```

Appendix 2

Example: Using a Reserved Word to Obtain an Opcode to Filter Events

The following code demonstrates the use of ESP_OPCODE to filter events. It uses a simple callback function that can be registered for a window's new event updates. The function receives the schema of the events passed to it and a set of one or more events bundled into a **dfESPeventblock** object.

For more information, see "Using Event Metadata in Expressions" on page 124.

```
// -*- Mode: C++; indent-tabs-mode: nil; c-basic-offset: 4 -*-

#define MAXROW 1024

// Include class definitions for  source windows, filter windows,
//    continous queries, and projects.
//
#include "dfESPwindow_source.h"
#include "dfESPwindow_filter.h"
#include "dfESPcontquery.h"
#include "dfESPproject.h"
#include "dfESPengine.h"

using namespace std;

void winSubscribeFunction(dfESPschema *os, dfESPeventblockPtr ob, void *ctx) {
    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
            dfESPengine::oStream() << buff << endl;  // print it
            if (ob->getData(i)->getOpcode() == dfESPeventcodes::eo_UPDATEBLOCK)
                ++i;  // skip the old record in the update block
        }  //for
    }  //if
}

// Test a filter window using ESP_OPCODE to filter out all but Inserts.
int main(int argc, char *argv[]) {

    // Call Initialize without overriding the framework defaults
    //    which for all paths & filenames will be relative to dirName,
    //    and for logging will be stdout.
    bool eventFailure;
    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_01;
project_01 = myEngine->newProject("project_01");

dfESPcontquery  *cq_01;
cq_01 = project_01->newContquery("contquery_01");

// Build the source window schema, source window, filter windows,
//    and continous query objects.
dfESPwindow_source *sw;
sw = cq_01->newWindow_source("sourceWindow_01", dfESPindextypes::pi_RBTREE,
     dfESPstring("ID*:int64,symbol:string,price:money,quant:
                      int32,vwap:double,trade_date:date,tstamp:stamp"));
dfESPschema  *schema_01 = sw->getSchema();

dfESPwindow_filter *fw;
fw = cq_01->newWindow_filter("filterWindow",
                                dfESPindextypes::pi_RBTREE);
fw->setFilter("ESP_OPCODE==\"I\"");


// Add the subscriber callback to the source window, and the
//     source window to the continous query.
fw->addSubscriberCallback(winSubscribeFunction);


cq_01->addEdge(sw, 0, fw);

project_01->setNumThreads(2);

myEngine->startProjects();

// declare some variables to build up the input data.
//
//
dfESPptrVect<dfESPeventPtr> trans;
dfESPevent   *p;

// Build a block of input data.
//
p = new dfESPevent(schema_01,(char *)
    "i,n,44001,ibm,101.45,5000,100.565,2010-09-07
     16:09:01,2010-09-07 16:09:01.123", eventFailure);
trans.push_back(p);
p = new dfESPevent(schema_01,(char *)
    "i,n,50000,sunw,23.52,100,26.3956,2010-09-08
     16:09:01,2010-09-08 16:09:01.123", eventFailure);
trans.push_back(p);
p = new dfESPevent(schema_01,(char *)
    "i,n,66666,orcl,120.54,2000,101.342,2010-09-09
     16:09:01,2010-09-09 16:09:01.123", eventFailure);
```

```
    trans.push_back(p);

    dfESPeventblockPtr ib =
      dfESPeventblock::newEventBlock(&trans, dfESPeventblock::ebt_TRANS);
    trans.free();

    // Put the event block into the graph, then loop over the graph until
    //    there is no more work to do.
    //
    project_01->injectData(cq_01, sw, ib);
    project_01->quiesce();  // quiesce the graph of events

    // Build another block of input data.
    p = new dfESPevent(schema_01,(char *)
        "u,n,44001,ibm,100.23,3000,100.544,2010-09-09
         16:09:01,2010-09-09 16:09:01.123", eventFailure);
    trans.push_back(p);
    p = new dfESPevent(schema_01,(char *)
        "u,n,50000,sunw,125.70,3333,122.3512,2010-09-07
         16:09:01,2010-09-07 16:09:01.123", eventFailure);
    trans.push_back(p);
    p = new dfESPevent(schema_01,(char *)
        "u,n,66666,orcl,99.11,954, 97.4612,2010-09-10
         16:09:01,2010-09-10 16:09:01.123", eventFailure);
    trans.push_back(p);

    ib =  dfESPeventblock::newEventBlock(&trans, dfESPeventblock::ebt_TRANS);
    trans.free();
    project_01->injectData(cq_01, sw, ib);
    project_01->quiesce();  // quiesce the graph of events

    // Build another block of input data.
    p = new dfESPevent(schema_01,(char *)
        "d,n,66666,orcl,99.11,954, 97.4612,2010-09-10
         16:09:01,2010-09-10 16:09:01.123", eventFailure);
    trans.push_back(p);

    ib =  dfESPeventblock::newEventBlock(&trans, dfESPeventblock::ebt_TRANS);
    trans.free();
    project_01->injectData(cq_01, sw, ib);
    project_01->quiesce();  // quiesce the graph of events

    // cleanup
    dfESPengine::shutdown();
    return 0;
}
```

Appendix 3

Example: Using DataFlux Expression Language Global Functions

The following code creates a compute window that uses a UDF in a compute expression for a string field. The function is initialized using the window-expression **`init`** feature.

```
// -*- 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 "dfESPcontquery.h"
#include "dfESPeventblock.h"
#include "dfESPproject.h"

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

#include <cstdio>
#include <iostream>

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[]) {

    bool eventFailure;
    // 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");

dfESPwindow_source *sw;
sw = contQuery->newWindow_source("sourceWindow",  dfESPindextypes::pi_HASH,
     dfESPstring("name:string,ID*:int32,city:string"));
dfESPschema  *sw_schema = sw->getSchema();

dfESPwindow_compute *cw;
cw = contQuery->newWindow_compute("computeWindow",  dfESPindextypes::pi_HASH,
     dfESPstring("ID*:int32,name:string,city:string,udfVal1:int32,udfVal2:int32"));

// Register a UDF expression for this window to be used in field calc expressions.
cw->regWindowExpUDF("return ((ID+3)*2)",
                    "example_udf1", dfESPdatavar::ESP_INT32);
cw->regWindowExpUDF("return ((ID+5)*3)",
                    "example_udf2", dfESPdatavar::ESP_INT32);
// 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
cw->addNonKeyFieldCalc("example_udf1()");  // call UDF to fill this field
cw->addNonKeyFieldCalc("example_udf2()");  // call UDF to fill this field

// 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", eventFailure);
trans.push_back(p);
p = new dfESPevent(sw_schema,(char *)"i,n,Scott, 1112, Cary", eventFailure);
trans.push_back(p);
p = new dfESPevent(sw_schema,(char *)"i,n,someone, 1113, Raleigh", eventFailure);
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;
```

```
}
```

The following code creates a source window that uses a splitter expression UDF to determine where it should send subsequent events. Recipients are one of two connected copy windows. One copy window gets events with even-numbered IDs. The other gets events with odd-numbered IDs.

```
// -*- Mode: C++; indent-tabs-mode: nil; c-basic-offset: 4 -*-

// Include class definitions for modeling objects.
//
#include "dfESPstring.h"
#include "dfESPevent.h"
#include "dfESPwindow_source.h"
#include "dfESPwindow_copy.h"
#include "dfESPcontquery.h"
#include "dfESPeventblock.h"
#include "dfESPengine.h"
#include "dfESPproject.h"

// Standard includes
#include <iostream>
#include <stdlib.h>
#include <cstdio>
#include <iostream>

using namespace std;

struct callback_ctx {
        dfESPthreadUtils::mutex *lock;
        dfESPstring windowName;
};

// This call back function is registered to the source and copy windows.
// It uses the context pointer to get the appropriate calling window name and
// to lock on output for thread safetyness.
//
void winSubscribe(dfESPschema *os, dfESPeventblockPtr ob, void *cx) {
        callback_ctx *ctx = (callback_ctx *)cx;

        ctx->lock->lock();
        dfESPengine::oStream()
        << endl << "-------------------------------------------------------" << endl;
        dfESPengine::oStream() << ctx->windowName << endl;
        ob->dump(os);
        ctx->lock->unlock();
}


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 it's output,
   //       -b <badevent.name> to write any bad events (events that failed
   //          to be applied to a window index).
   //       -r <restore.path> path used to restore a previously persisted
   //          engine state.
   // @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: dfESPLLInfo,
   //        @see dfESPLoggingLevel
   // @param logConfigFile a log4SAS configuration file
   //        - default: configure logging to go to standard out.
   // @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");

   // 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
   //
   dfESPwindow_source *sw;
   sw = cq_01->newWindow_source("source",  dfESPindextypes::pi_RBTREE,
       dfESPstring("ID*:int32,symbol:string,price:double"));

   // Register the User Defined Expression with window splitter's expression
       // engine.   This UDF does a mod 2 on the ID field, so either slot 0 or
       // slot 1 will be selected for each event.
   //
   sw->regSplitterExpUDF("return ID%2", "example_udf", dfESPdatavar::ESP_INT32);

   // Use the setSplitter call to set the splitter expression which uses
```

```
    // the user defined function already registered.
    //
sw->setSplitter("example_udf()");

// Create the copy windows.
dfESPwindow_copy *cw_even;
cw_even = cq_01->newWindow_copy("copy_even",  dfESPindextypes::pi_RBTREE);

dfESPwindow_copy *cw_odd;
cw_odd = cq_01->newWindow_copy("copy_odd",  dfESPindextypes::pi_RBTREE);

// Add the subscriber callbacks to the source & copy windows
    //    using context data structures for each
    //
    callback_ctx src_ctx, cpy_even_ctx, cpy_odd_ctx;

    src_ctx.lock = cpy_even_ctx.lock = cpy_odd_ctx.lock =
                   dfESPthreadUtils::mutex::mutex_create(); // a shared lock
    src_ctx.windowName = "source";  // window name for callback function
    cpy_even_ctx.windowName = "copy_even";  // window name for callback function
    cpy_odd_ctx.windowName = "copy_odd";  // window name for callback function
sw->addSubscriberCallback(winSubscribe, (void *)&src_ctx);
cw_even->addSubscriberCallback(winSubscribe, (void *)&cpy_even_ctx);
cw_odd->addSubscriberCallback(winSubscribe, (void *)&cpy_odd_ctx);

// Add the connectivity information to the continuous query. This
//    means sw[slot 0] --> cw_even
//          sw[slot 1] --> cw_odd
//
cq_01->addEdge(sw, 0,  cw_even);
cq_01->addEdge(sw, 1,  cw_odd);

// 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(3);
myEngine->startProjects();

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

/* Now build some test data and inject it into the source window. */

bool eventFailure;
dfESPptrVect<dfESPeventPtr> trans;
dfESPevent   *p;

// Build a block of input data.
//
p = new dfESPevent(sw->getSchema(),(char *)"i,n,1,ibm,101.45", eventFailure);
trans.push_back(p);
p = new dfESPevent(sw->getSchema(),(char *)"i,n,2,sunw,23.5", eventFailure);
```

```
    trans.push_back(p);
    p = new dfESPevent(sw->getSchema(),(char *)"i,n,3,orcl,10.1", eventFailure);
    trans.push_back(p);

    dfESPeventblockPtr ib =
   dfESPeventblock::newEventBlock(&trans, dfESPeventblock::ebt_TRANS);
    trans.free();

    // Put the event block into the graph, then loop over the graph until
    //    there is no more work to do.
    //
    project_01->injectData(cq_01, sw, ib);

    // Quiesce the project to ensure all events are processed before shuting down.
    project_01->quiesce();

    // Cleanup.
    myEngine->shutdown();  // Shutdown the ESP engine
    return 0;
}
```

Appendix 4

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 expressions. To use these functions, you must have installed the SAS DataFlux QKB (Quality Knowledge Base) product and set two environment variables: **DFESP_QKB** and **DFESP_QKB_LIC**. For more information, see "Using Blue Fusion Functions" on page 126.

```
// -*- 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 "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

// This example creates a compute window that contains a field
// expression that uses a data quality function in the Blue Fusion library.
// In this example we are standardizing the city name.
//
// In order to run this example, you need to download the DataFlux Quality
// Knowledge Base and set the environment variable DFESP_QKB to the root node
// of that install.

using namespace std;


void winSubscribe_compute(dfESPschema *os, dfESPeventblockPtr ob, void *ctx) {
    dfESPengine::oStream() << endl
```

```
<< "-----------------------------------------------------------------" << endl;
    dfESPengine::oStream() << "computeWindow" << endl;
    ob->dump(os);
    }

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

    bool eventFailure;
    // 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 source window schema, source window, copy window,
    //    and continous query objects.

    dfESPwindow_source *sw;
    sw = contQuery->newWindow_source("sourceWindow", dfESPindextypes::pi_HASH,
        dfESPstring("name:string,ID*:int32,city:string"));
    dfESPschema  *sw_schema = sw->getSchema();

    dfESPwindow_compute *cw;
    cw = contQuery->newWindow_compute("computeWindow", 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

    // Now run city through the blue fusion standardize function.
    char newCity[2048] = "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", eventFailure);
trans.push_back(p);
p = new dfESPevent(sw_schema,(char *)"i,n,Scott, 1112, caryy", eventFailure);
trans.push_back(p);
p = new dfESPevent(sw_schema,(char *)"i,n,someone, 1113, rallleigh", eventFailure);
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 5

Setting the Logging Level

You can set logging levels for the engine, adapters, and the XML Modeling Server. The valid logging levels are as follows:

Logging Level	Description
TRACE	Provides detailed information.
DEBUG	Provides detailed information about the flow through the system. (Provides more information than TRACE.)
INFO	Provides information about run-time events of interest. (Provides less information than TRACE.)
WARN	Shows warning messages (for example, use of deprecated APIs, poor use of an API, or other run-time situations that are undesirable).
ERROR	Shows all error messages.
FATAL	Shows message for fatal errors.
OFF	Specifies that logging is turned off.

You can set the logging levels for the engine using the **`dfESPengine::initialize()`** method or through a log configuration file that can also be specified in this same method. To set the logging level for the C publish/subscribe API, use the **`C_dfESPpubsubInit()`** method. To set the logging level for the Java publish/subscribe API, use the **`init()`** method. Connectors use engine logging.

Logging levels for adapters use the same range of levels.

Table A5.1 *Logging Level for the Adapter*

Logging Level	Parameter Setting	Location
TRACE	dfESPLLTrace	Logs
DEBUG	dfESPLLDebug	Logs
INFO	dfESPLLInfo	Immediately visible on the console.
WARN	dfESPLLWarn	Immediately visible on the console.

Logging Level	Parameter Setting	Location
ERROR	dfESPLLError	Immediately visible on the console.
FATAL	dfESPLLFatal	Immediately visible on the console.
OFF	dfESPLLOff	Not applicable.

The XML Modeling Server provides logging contexts.

Logging Context	Description
esp	Represents the basic engine and generates messages about events that flow through the system, throttling, system errors, and so on.
common.http	Details HTTP requests that come into the server and their corresponding responses.
common.smtp	Details SMTP server interaction in notification windows.
esp.http	Details event-stream-processing specific HTTP requests.
esp.pubsub	Details event-stream-processing publish/subscribe activity.
esp.server	Provides information about general XML Modeling Server issues.
esp.windows	Provides information about window issues.
pubsub.http	Details requests and responses from the publish/subscribe HTTP server.
window.notification	Shows messages specific to notification windows.

The **esp** context is always available.

To set the start-up log level for the XML Modeling Server, use the **loglevel** parameter on the command line.

For example, enter **dfesp_xml_server -loglevel "esp=error,common.http=debug,esp.windows=trace"**.

This sets the log level for each specified context before the server starts to perform its basic functions.

If you start the XML Modeling Server through an HTTP administrative interface, you can both query and set the log level for active logging contexts. Do this with the **/loggers** and **/loglevel** requests, respectively.

To query the current log levels, the request is as follows: **http://esp-base:46001/loggers**.

The response to this request is as follows:

```
<loggers>
```

```
   <logger name='esp' level='ERROR'/>
   <logger name='common.http' level='INFO'/>
   <logger name='esp.http' level='INFO'/>
   <logger name='esp.pubsub' level='INFO'/>
   <logger name='esp.server' level='INFO'/>
   <logger name='esp.windows' level='INFO'/>
   <logger name='pubsub.http' level='INFO'/>
   <logger name='window.notification' level='INFO'/>
</loggers>
```

When you are interested only in certain contexts, use the **contexts** parameter: **http://esp-base:46001/loggers?contexts=esp,esp.http**.

The response to this request is as follows:

```
<loggers>
   <logger name='esp' level='ERROR'/>
   <logger name='esp.http' level='INFO'/>
</loggers>
```

To set the log level for a particular context, use the **/loglevel** request: **http://esp-base:46001/loglevel/esp/trace**.

The response to this request is as follows:

```
<response>
    <message>logger 'esp' set to 'trace'</message>
</response>
```

To set the log level for all contexts, use the **/loglevel** request: **http://esp-base:46001/loglevel/*/trace**.

The response to this request is as follows:

```
<response>
    <message>all loggers set to 'trace'</message>
</response>
```

Recommended Reading

SAS Event Stream Processing is supported by the following documents:

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

  Specifically, documentation about the following topics is provided:

  - C++ Modeling API
    - SAS Event Stream Processing API
    - SAS Event Stream Processing Connector API
  - Publish/Subscribe API
    - SAS Event Stream Processing Publish/Subscribe C API
    - SAS Event Stream Processing Publish/Subscribe Java API
- View **`$DFESP_HOME/etc/xml/schema/model.rnc`** for a full syntactic description of valid SAS Event Stream Processing XML language elements.

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

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Glossary

derived windows
windows that display events that have been fed through other windows and that perform computations or transformations on these incoming events.

directed graph
a set of nodes connected by edges, where the edges have a direction associated with them.

engine
the top-level container in a model that manages the project resources.

event block
a grouping or package of events with a unique ID for use in a continuous query.

event stream
a continuous flow of event blocks.

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

factory server
a server for factory objects that control the creation of other objects, access to other objects, or both.

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

modeling API
an application programming interface that enables developers to write event stream processing models.

operation code (opcode)
an instruction that specifies an action to be performed.

publish/subscribe API
a library that enables you to publish event streams into an event stream processor, or to subscribe to event streams, within the event stream processing model. The publish/subscribe API also includes a C and JAVA event stream processing object support library.

source window
a window that has no windows feeding into it and is the entry point for publishing events into the continuous query.

stream
a sequence of data elements [that are] made available over time.

thread pool
a set of threads that can be used to execute tasks, post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.

window
a processing node in an event stream processing model. Source and derived windows can perform aggregations, computations, pattern matching, and other operations.