



THE
POWER
TO KNOW.

SAS[®] Event Stream Processing 3.1

Overview

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

SAS® Event Stream Processing 3.1: Overview

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

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

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

Contents

<i>Using This Book</i>	v
Chapter 1 • Overview to SAS Event Stream Processing	1
Product Overview	1
Chapter 2 • Example: Running a Continuous Query	3
Overview of the Example	4
Processing Events	5
C++ Code to Implement the Example	9
Building and Running the Source Code	16
Chapter 3 • Example: Processing Trades	19
Overview of the Example	19
C++ Code for Processing Trades	20
XML Code for Processing Trades	22
Running the XML Code	25
 <i>Recommended Reading</i>	 27
<i>Glossary</i>	29

Using This Book

Audience

This book provides an overview to SAS Event Stream Processing and describes its basic functionality. It also provides step-by-step examples that you can adapt for personal use. The intended audience is assumed to be new to the product and possibly new to SAS.

Overview to SAS Event Stream Processing

<i>Product Overview</i>	1
-------------------------------	---

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.

2

Example: Running a Continuous Query

<i>Overview of the Example</i>	4
<i>Processing Events</i>	5
Processing the First Event	5
Processing the Second Event	6
Processing the Third Event	7
Processing the Fourth Event	7
Processing the Fifth Event	9
<i>C++ Code to Implement the Example</i>	9
Overview to the Code	9
Create Callback Functions	10
Create the Engine	11
Create a Project	11
Define a Continuous Query Object	12
Build the Source and Filter Windows	12
Register the Filter Expression and Add Connectivity Information	13
Start the Project and Inject Data	13
Build Input Data and Insert Events	14
Clean Up and Shut Down Services	16
<i>Building and Running the Source Code</i>	16
How to Build and Run the Source Code	16
Build Results	17

Overview of the Example

The following example passes events through a source window and then a single filter window. Events conform to a proscribed *schema*. The schema is a structured string that defines and specifies the order of a set of variables in an event.

The following processing steps are demonstrated:

- running a simple continuous query on a published event stream
- performing a filtering computation
- determining specific events to produce in each step of processing

Here is the schema of the source window:

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

The filter window inherits this schema from the source window.

The schema consists of four fields:

Field	Type
ID	signed 32-bit integer
symbol	literal constant
quantity	signed 32-bit integer
price	double precision floating-point

The ID field has the * designator to indicate that this field is part of the key for the window. No other field has this designator, so the ID field completely forms the key.

Key fields are used to identify an event for operations such as Insert, Update, Delete, or Upsert. Key fields must be unique for an event. You can think of the event stream as a database and the key fields as lookup keys.

A filter expression `quantity > 1000` specifies that events are to be passed through the filter only when the `quantity` field in the event exceeds the value of 1000.

Events that enter a source window must have an operation code (opcode). The opcode can be Insert (I), Update (U), Delete (D), or Upsert (P).

Opcode	Description
Insert (I)	Adds event data to a window.
Update (U)	Changes event data in a window.
Delete (D)	Removes event data from a window.
Upsert (P)	A merge function in which data for an event is updated, inserted, or both.

In the following sections, assume that an application feeds five events into the source window. The lifecycle of events is traced through the continuous query. How to run this application is described in [“Building and Running the Source Code” on page 16](#).

Processing Events

Processing the First Event

The first event is as follows:

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

- 1 The source window receives `e1` as an Input event. It stores the event and passes it to the filter window.
- 2 The filter window receives `e1` as an Input event, as designated by the “i” in the first field. The second field in this and all subsequent events designates “normal.”
- 3 The `quantity` field has a value of 2000. Because the filter expression is `quantity > 1000`, the filter window stores the input. Typically, a filter window would pass `e1`

forward. However, because the filter window has no dependent windows, there is no additional data flow for the event.

The window contents are now as follows:

Source Window				Filter Window			
ID	Symbol	Qty	Price	ID	Symbol	Qty	Price
10	IBM	2000	164.1	10	IBM	2000	164.1

Processing the Second Event

The second event is as follows:

e_2 : [p,n,20,MS,1000,26.67]

- 1 The source window receives e_2 as an Upsert event. It checks whether the window has a stored event with a key (ID) of 20.
- 2 An ID of 20 is not stored, so the source window creates a new event e_{2a} : [I, 20, "MS", 1000, 26.67]. It stores this new event and passes it to the filter window.
- 3 The filter window receives e_{2a} as an Input event.
- 4 The value in the **quantity** field of e_2 equals 1000, which does not meet the condition set by the filter expression in the schema. Thus, this event is not stored or passed to any dependent windows.

The window contents are now as follows:

Source Window				Filter Window			
ID	Symbol	Qty	Price	ID	Symbol	Qty	Price
10	IBM	2000	164.1	10	IBM	2000	164.1
20	MS	1000	26.67				

Processing the Third Event

The third event is as follows:

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

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

- 1 The source window receives `e3` as a Delete event.
- 2 The source window looks up the event that is stored with the same key. The Delete opcode removes the event from the source window.
- 3 The source window passes the found record to the filter window with the Delete opcode specified. In this case, the record that is passed to the filter window is as follows:


```
e3a: [d,n,10,IBM,2000,164.1]
```
- 4 The filter window receives `e3a` as an Input event.
- 5 The value in the `Quantity` field of `e3a` equals 2000. This old event that was previously stored makes it through the filter, so it is removed.

The window contents are now as follows:

Source Window				Filter Window			
ID	Symbol	Qty	Price	ID	Symbol	Qty	Price
20	MS	1000	26.67				

Processing the Fourth Event

The fourth event is as follows:

```
e4: [u,n,20,MS,3000,26.99]
```

- 1 The source window receives e_4 as an Update event.
- 2 The source window looks up the event stored with the same key and modifies it.
- 3 The source window constructs an update block that consists of the new record with updated values marked as an update block followed by the old record that was updated.
- 4 The block is marked as a Delete event. The new event Update block that is passed to the filter window looks like this:

e_{4a} : [ub,n,20,MS,3000,26.99] , [d,n,20,MS,1000,26.67]

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

- 5 The filter window receives e_{4a} as an Input event.
 - 6 The value in the **quantity** field of $e_{4a} > 1000$, but previously it was ≤ 1000 . The input did not pass the previous filter condition, but now it does pass. Because the input is not present in the filter window, the filter window generates an Insert event of the following form:
- e_{4b} : [i,n,20,MS,3000,26.99]
- 7 The Insert event is stored. The filter window would pass e_{4b} . However, because there are no dependent windows, this input does not pass. There is no further data flow for this event.

The window contents are now as follows:

Source Window				Filter Window			
ID	Symbol	Qty	Price	ID	Symbol	Qty	Price
20	MS	3000	26.99	20	MS	3000	26.99

Processing the Fifth Event

The fifth event is as follows:

```
e5: [i,n,30,ACL,2000,2.11]
```

- 1 The source window receives `e5` as an Insert event, stores it, and passes `e1` to the filter window.
- 2 The filter window receives `e5` as an Input event. Because the value in the `quantity` field > 1000, the filter window stores the input. Because the filter window has no dependent windows, there is no further data flow.

The window contents are now as follows:

Source Window				Filter Window			
ID	Symbol	Qty	Price	ID	Symbol	Qty	Price
20	MS	3000	26.99	20	MS	3000	26.99
30	ACL	2000	2.11	30	ACL	2000	2.11

C++ Code to Implement the Example

Overview to the Code

The following C++ code implements the example. You can find this code in `$DFESP_HOME/src/filter_exp`. Edit the associated Makefile to remove the comments for architecture-specific build variables.

Create Callback Functions

Before you create the functions that process events, include header files provided with the SAS Event Stream Processing modeling and execution library (for example, `dfESPEventblock.h`). Then declare two callback functions, one for the source window and the other for the filter window. You register these functions with the source window and the filter window to print the events that these windows generate.

You can choose to define callback functions each window. In this example, one callback function is registered with the source window and another function is registered for the filter window. The callback for the source window receives the schema of the events that it receives and a set of one or more events bundled into a `dfESPEventblock` object. It also has an optional context pointer to share state across calls or to pass state into calls. The callback function for the filter window is identical to that for the source window. The callback functions are used to send a message to standard output.

```
// -*- Mode: C++; indent-tabs-mode: nil; c-basic-offset: 4 -*-
//
#include "dfESPEventblock.h"
#include "dfESPEvent.h"
#include "dfESPwindow_source.h"
#include "dfESPwindow_filter.h"
#include "dfESPcontquery.h"
#include "dfESPengine.h"
#include "dfESPproject.h"

using namespace std;

// Callback function for the source window.
//
void winSubscribe(dfESPschema *os, dfESPEventblockPtr ob, void *context) {
    callback_ctx *ctx = (callback_ctx *)context;

    ctx->lock->lock();
    cerr << endl << "-----" << endl;
    cerr << ctx->windowName << endl;

    // the dfESPEventblock has a dump() method that prints each
    // event that the event block contains.
    ob->dump(os);
    ctx->lock->unlock();
}
```


Create the Engine

Create the engine that sets up fundamental services such as licensing, logging, publish/subscribe, and threading.

```
// Main program - closing bracket appears at the very end of the code block
int main(int argc, char *argv[]) {

// ----- BEGIN MODEL (CONTINUOUS QUERY DEFINITIONS) -----
// @param argc the parameter count as passed into main.
// @param argv the paramter vector as passed into main. Currently
//     the dfESP only looks for -t <textfile.name> to write its
//     output and -b <badevent.name> to write any bad events (events
//     that failed to be applied to a window index).
// @param id the user supplied name of the engine.
// @param pubsub pub/sub enabled or disabled and port pair,
//     formed by calling static function dfESPengine::pubsubServer().
// @param logLevel the lower threshold for displayed log messages -
//     default: dfESPLLTrace, @see dfESPLoggingLevel
// @param logConfigFile a logging facility configuration file
//     - default: stdout.
// @param licKeyFile a FQPN to a license file -
//     default: $DFESP_HOME/etc/license/esp.lic
// @return the dfESPengine instance.
//
dfESPengine *myEngine = dfESPengine::initialize(argc, argv,
    "engine", pubsub_DISABLE);
    if (myEngine == NULL) {
        cerr <<"Error: dfESPengine::initialize() failed using all
            framework defaults\n";
        return 1;
    }
}
```

Create a Project

Ordinarily, engines contain one or more projects. Define the project.

```
// Define the project
dfESPproject *project_01 = myEngine->newProject("project_01");
```

Define a Continuous Query Object

Typically, projects contain one or more continuous queries. Define a continuous query object. This is the first-level container for source and derived windows. The object also contains window-to-window connectivity information.

```
// Create a continuous query
dfESPcontquery *cq_01;
cq_01 = project_01->newContquery("contquery_01");
```

Build the Source and Filter Windows

Build the source window. Specify the following:

- the window name.
- the schema for events.
- the depot used to generate the index and to handle event storage.
- the type of primary index, which defines how event indexing occurs. In this case, the primary index is a hash tree.

```
// Build the source window
dfESPwindow_source *sw_01;
sw_01 = cq_01->newWindow_source("sourceWindow_01",
dfESPindextypes::pi_HASH,
dfESPstring("ID*:int64,symbol:string,price:money,quant:int32,
vwap:double,trade_date:date,tstamp:stamp"));
```

Next, build the filter window. Specify the object name, the depot used to generate the index and to handle event storage, and the type of primary index. In this case, the primary index is a hash tree. Unlike with the source window, you do not need to specify the schema. The filter window uses the same schema as the window that provided input to it.

```
// Build a filter window
dfESPwindow_filter *fw_01;
fw_01 = cq_01->newWindow_filter("filterWindow_01",
dfESPindextypes::pi_HASH);
fw_01->setFilter("quant>1000");
```

Register the Filter Expression and Add Connectivity Information

Register the filter expression (quant>1000) for this window. Add the subscriber callback to the source and filter windows. These functions are called whenever a window produces output events. The events produced are both passed to these callback functions, and also sent farther down the directed graph for additional processing. Here, you format the events as CSV rows and dump them to your display. This enables you to see what each window produces at each step of the computation.

```
callback_ctx src_ctx, filter_ctx;
    src_ctx.lock = filter_ctx.lock =
        dfESPthreadUtils::mutex::mutex_create(); // a shared lock
    src_ctx.windowName = "sourceWindow_01"; // window name for callback function
    filter_ctx.windowName = "filterWindow_01"; // window name for callback function
    sw_01->addSubscriberCallback(winSubscribe, (void *)&src_ctx);
    fw_01->addSubscriberCallback(winSubscribe, (void *)&filter_ctx);
```

Add the connectivity information to the continuous query. In this case, connect sw_01[slot 0] to fw_01.

```
cq_01->addEdge(sw_01, fw_01);
```

Start the Project and Inject Data

Define the project's thread pool size and start it. After you start the project, you do not see anything happen because no data has yet been put into the continuous query.

```
project_01->setNumThreads(1);
myEngine->startProjects();

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

At this point, the project is running in the background using the defined thread pool. Use the main thread to inject data. In production applications, you might dedicate a thread for each active source window input event stream to optimize performance.

```
cerr <<endl <<endl;

// Declare some scratch variables to build up and submit the input
// data.
```

14 Chapter 2 / Example: Running a Continuous Query

```
//
dfESPptrVect<dfESPEventPtr> trans;
dfESPEvent *p;
dfESPEventblockPtr ib;
bool eventFailure;

//
// ----- BEGIN - DEFINE BLOCKS OF EVENTS AND INJECT into
//   running PROJECT -----
//

cout <<endl<<endl; // Logging uses cout as well, so just use white space
for events.
```

Build Input Data and Insert Events

Build a block of input data with three insert events. Typically, events are generated by one or more publishing applications.

```
// dfESPEvent() takes the event schema and the event character string
//   where: the i is insert
//   else {u|p|d} mean update, upsert and delete respectively.
//   The n is normal.
//   The rest are the field values for the event.
//

p = new dfESPEvent(sw_01->getSchema(),
    (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(sw_01->getSchema(),
    (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(sw_01->getSchema(),
    (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);

ib = dfESPEventblock::newEventBlock(&trans, dfESPEventblock::ebt_TRANS);
trans.free(); // this clears the vector and frees memory
```

Inject the event block into the graph. Typically, you use the Publish and Subscribe API to subscribe to events published locally or on a networked computer system. The following `injectdata` call is a way to bypass the API and can be useful for testing.

The `injectdata` call is asynchronous with respect to processing. It deposits the input block into the queue of the source window, and then the thread pool takes over. Given this, use `quiesce()` to stop the thread until all the injected events have been processed through the entire continuous query.

```
project_01->injectData(cq_01, sw_01, ib);
project_01->quiesce(); // quiesce the graph of events

// Build & inject another block of input events, this time with updates.
//
p = new dfESPevent(sw_01->getSchema(),
    (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(sw_01->getSchema(),
    (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(sw_01->getSchema(),
    (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_01, ib);
project_01->quiesce(); // quiesce the graph of events
```

Build and inject another block, this time with a single delete event.

```
p = new dfESPevent(sw_01->getSchema(),
    (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_01, ib);
project_01->quiesce(); // quiesce the graph of events
```

```
cout <<endl<<endl; // Logging uses cout as well, so just use white
space for events

//
// ----- END - DEFINE BLOCKS OF EVENTS AND INJECT into running
// PROJECT -----
```

Clean Up and Shut Down Services

Finally, clean up and shut down services.

```
myEngine->shutdown();
return 0;
}
```

Building and Running the Source Code

How to Build and Run the Source Code

Suppose that the SAS Event Stream Processing library is installed in `/opt/dfESP`. You would enter these settings:

```
export DFESP_HOME = /opt/dfESP
export LD_LIBRARY_PATH = $DFESP_HOME/lib
```

A Perl script, `$DFESP_HOME/bin/dfespenv`, sets these environment variables. However, you can also add these settings to your login shell or script.

Navigate to the example directory, which is `$DFESP_HOME/src/filter_exp`. Use the `make` command to build the example. In the Makefile for `filter_exp`, you find the following comments for the GNU Compiler Collection (GCC) on Linux:

```
# -- GCC on Linux
#   Uncomment the next three lines to use these settings.
# CXX=g++
# CXXFLAGS=-g -m64
# LDFLAGS=-L$DFESP_HOME/lib
```

Build Results

Building the code creates an executable file that you can run. What follows depicts the results of running that executable.

```
-----
sourceWindow_01
-----
```

```
TID: 0x0005000000000001
```

```
depth: 1
```

```
event [0]: <I,N: 44001,ibm,101.45,5000,100.565000,2010-09-07
16:09:01,2010-09-07 16:09:01.123000>
event [1]: <I,N: 50000,sunw,23.52,100,26.395600,2010-09-08
16:09:01,2010-09-08 16:09:01.123000>
event [2]: <I,N: 66666,orcl,120.54,2000,101.342000,2010-09-09
16:09:01,2010-09-09 16:09:01.123000>
```

```
-----
filterWindow_01
-----
```

```
TID: 0x0005000000000001
```

```
depth: 2
```

```
event [0]: <I,N: 44001,ibm,101.45,5000,100.565000,2010-09-07
16:09:01,2010-09-07 16:09:01.123000>
event [1]: <I,N: 66666,orcl,120.54,2000,101.342000,2010-09-09
16:09:01,2010-09-09 16:09:01.123000>
```

```
-----
sourceWindow_01
-----
```

```
TID: 0x0005000000000002
```

```
depth: 1
```

```
event [0]: <UB,N: 44001,ibm,100.23,3000,100.544000,2010-09-09
16:09:01,2010-09-09 16:09:01.123000>
event [1]: <D,N: 44001,ibm,101.45,5000,100.565000,2010-09-07
16:09:01,2010-09-07 16:09:01.123000>
event [2]: <UB,N: 50000,sunw,125.7,3333,122.351200,2010-09-07
16:09:01,2010-09-07 16:09:01.123000>
event [3]: <D,N: 50000,sunw,23.52,100,26.395600,2010-09-08
16:09:01,2010-09-08 16:09:01.123000>
event [4]: <UB,N: 66666,orcl,99.11,954,97.461200,2010-09-10
16:09:01,2010-09-10 16:09:01.123000>
event [5]: <D,N: 66666,orcl,120.54,2000,101.342000,2010-09-09
16:09:01,2010-09-09 16:09:01.123000>
```

```
-----
filterWindow_01
-----
```

18 Chapter 2 / Example: Running a Continuous Query

TID: 0x0005000000000002

depth: 2

event [0]: <UB,N: 44001,ibm,100.23,3000,100.544000,2010-09-09

16:09:01,2010-09-09 16:09:01.123000>

event [1]: <D,N: 44001,ibm,101.45,5000,100.565000,2010-09-07

16:09:01,2010-09-07 16:09:01.123000>

event [2]: <I,N: 50000,sunw,125.7,3333,122.351200,2010-09-07

16:09:01,2010-09-07 16:09:01.123000>

event [3]: <D,N: 66666,orcl,120.54,2000,101.342000,2010-09-09

16:09:01,2010-09-09 16:09:01.123000>

sourceWindow_01

TID: 0x0005000000000003

depth: 1

event [0]: <D,N: 66666,orcl,99.11,954,97.461200,2010-09-10

16:09:01,2010-09-10 16:09:01.123000>

3

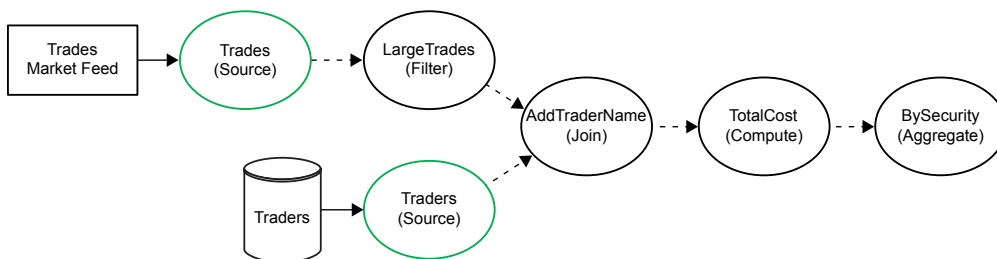
Example: Processing Trades

<i>Overview of the Example</i>	19
<i>C++ Code for Processing Trades</i>	20
<i>XML Code for Processing Trades</i>	22
<i>Running the XML Code</i>	25

Overview of the Example

Consider the following continuous query.

Figure 3.1 *Continuous Query Diagram*



In this continuous query, there are two source windows:

- the Trades window streams data about securities transactions from a trades market feed

- The Traders window streams data about who performs those transactions. This data could be published from a file, a database, or some other source.

As the source windows get data, the following occurs:

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

C++ Code for Processing Trades

The following C++ program implements the model. The program follows these steps.

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

8 Define the AddSecurity aggregate window.

9 Set the number of threads for the engine. Run the project. Shut down the engine.

```
#include "dfESPEngine.h"
#include "dfESPwindow_source.h"
#include "dfESPwindow_aggregate.h"
#include "dfESPwindow_filter.h"
#include "dfESPwindow_join.h"
#include "dfESPwindow_compute.h"
#include "dfESPcontquery.h"
#include "dfESPproject.h"

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

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

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

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

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

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

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

```

dfESPwindow_compute      *TotalCost
=cq_01->newWindow_compute("TotalCost",
                          dfESPindextypes::pi_RBTREE,
dfESPstring("tradeID*:string,security:string,quantity:int32,
            price:double,totalCost:double,traderID:int64,time:stamp,
            name:string"));
TotalCost->addNonKeyFieldCalc("security");
TotalCost->addNonKeyFieldCalc("quantity");
TotalCost->addNonKeyFieldCalc("price");
TotalCost->addNonKeyFieldCalc("price*quantity");
TotalCost->addNonKeyFieldCalc("traderID");
TotalCost->addNonKeyFieldCalc("time");
TotalCost->addNonKeyFieldCalc("name");

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

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

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

dfESPengine::shutdown();

return 0;
}

```

XML Code for Processing Trades

The following code renders the model in the SAS Event Stream Processing XML modeling language:

```

<engine port='55555' dateformat='%d/%b/%Y:%H:%M:%S'>
  <projects>

```

```

<project name='trades_proj' pubsub='auto' threads='4'>
  <contqueries>
    <contquery name='trades_cq'>
      <windows>
        <window-source name='Trades'
          index='pi_HASH'>
          <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-source name='Traders'>
          <schema>
            <fields>
              <field name='ID' type='int64' key='true' />
              <field name='name' type='string' />
            </fields>
          </schema>
        </window-source>

        <window-filter name='LargeTrades'>
          <expression>quantity >= 100</expression>
        </window-filter>

        <window-join name='AddTraderName'>
          <join type="leftouter">
            <conditions>
              <fields left='traderID' right='ID' />
            </conditions>
          </join>
          <output>
            <field-selection name='security' source='l_security' />
            <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='name' source='r_name' />
          </output>
        </window-join>

```

```

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

<window-aggregate name='BySecurity'>
  <schema>
    <fields>
      <field name='security' type='string' key='true' />
      <field name='quantityTotal' type='double' />
      <field name='costTotal' type='double' />
    </fields>
  </schema>
  <output>
    <field-expr>ESP_aSum(quantity)</field-expr>
    <field-expr>ESP_aSum(totalCost)</field-expr>
  </output>
</window-aggregate>
</windows>

<edges>
  <!-- Traders -> AddTraderName -->
  <edge source='LargeTrades' target='AddTraderName' />
  <edge source='Traders' target='AddTraderName' />

  <!-- Trades -> LargeTrades -> AddTraderName -> TotalCost -> BySecurity -->

```

```

        <edge source='Trades' target='LargeTrades' />
        <edge source='AddTraderName' target='TotalCost' />
        <edge source='TotalCost' target='BySecurity' />
    </edges>
</contquery>
</contqueries>
</project>
</projects>
</engine>

```

Running the XML Code

Here are the steps to run the XML code on Unix-like platforms.

- 1 Execute the event streams processing XML factory server with the model:

```

$DFESP_HOME/bin/dfesp_xml_server -model file://
full_path_to_xmlfile -http-admin 61001 -http-pubsub 61002.

```

- 2 Use `dfesp_fs_adapter` to populate the traders window with the events in the `traders.csv` file. The `traders.csv` file contains the input events for the Traders window.

```

$DFESP_HOME/bin/dfesp_fs_adapter -k pub -h dfESP://localhost:
55555/trades_proj/trades_cq/Traders -f traders.csv -t csv -b
256.

```

- 3 Use the Streamviewer tool to subscribe to the final BySecurity window to see the computed data as the trades data flows through the model. For more information about Streamviewer, see the *SAS Event Stream Processing: User's Guide*.

- 4 Use `dfesp_fs_adapter` to publish trades data from the `trades.csv` file. The `trades.csv` file contains the input events for the Trades window.

```

$DFESP_HOME/bin/dfesp_fs_adapter -k pub -h dfESP://localhost:
55555/trades_proj/trades_cq/Trades -f traders.csv -t csv -b 256
-d %d/%b/%Y:%H:%M:%S

```

Here are the steps to run the XML code on a Microsoft Windows 64 platform.

- 1 Execute the event stream processing XML factory server with the trades model:

```
%DFESP_HOME%\bin\dfesp_xml_server.exe -model file://  
full_path_to_trades -http-admin 61001 -http-pubsub 61002
```

- 2 Use `dfesp_fs_adapter` to populate the traders window with the events in the `traders.csv` file. The `traders.csv` file contains the input events for the Traders window.

```
%DFESP_HOME%\bin\dfesp_fs_adapter.exe -k pub -h dfESP://  
localhost:55555/trades_proj/trades_cq/Traders -f traders.csv -  
t csv -b 256
```

- 3 Use the Streamviewer tool to subscribe to the final BySecurity window to see the computed data as the trades data flows through the model. For more information about Streamviewer, see the *SAS Event Stream Processing: User's Guide*.

- 4 Use `dfesp_fs_adapter` to publish trades data from the `trades.csv` file. The `trades.csv` file contains the input events for the Trades window.

```
%DFESP_HOME%\bin\dfesp_fs_adapter.exe -k pub -h dfESP://  
localhost:55555/trades_proj/trades_cq/Trades -f traders.csv -t  
csv -b 256 -d %d/%b/%Y:%H:%M:%S
```


Recommended Reading

SAS Event Stream Processing is supported by the following documents:

- *SAS Event Stream 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 application programs.
- Open `$DFESP_HOME/doc/html/index.html` in a web browser to access detailed class and method documentation for the C++ modeling, C, and Java™ client publish/subscribe APIs. The documentation is organized by modules, namespaces, and classes.

Specifically, documentation about the following topics is provided:

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

For a complete list of SAS 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

28 *Recommended Reading*

Cary, NC 27513-2414

Phone: 1-800-727-0025

Fax: 1-919-677-4444

E-mail: 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.