SAS® Event Stream Processing 4.2: Examples

Running a Continuous Query

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

Field	Type
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 9.

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:

```
e2: [p,n,20,MS,1000,26.67]
```

1. The source window receives `e2` 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 `e2a: [I, 20, "MS", 1000, 26.67]`. It stores this new event and passes it to the filter window.
3. The filter window receives `e2a` as an Input event.
4. The value in the **`Quantity`** field of `e2` 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 `e4` 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:

   ```
   e4a: [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 `e4a` as an Input event.
6. The value in the **`Quantity`** field of `e4a` > 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:

   ```
   e4b: [i,n,20,MS,3000,26.99]
   ```
7. The Insert event is stored. The filter window would pass `e4b`. 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

Source Window				Filter Window			
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/examples/cxx/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.
//
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>
```

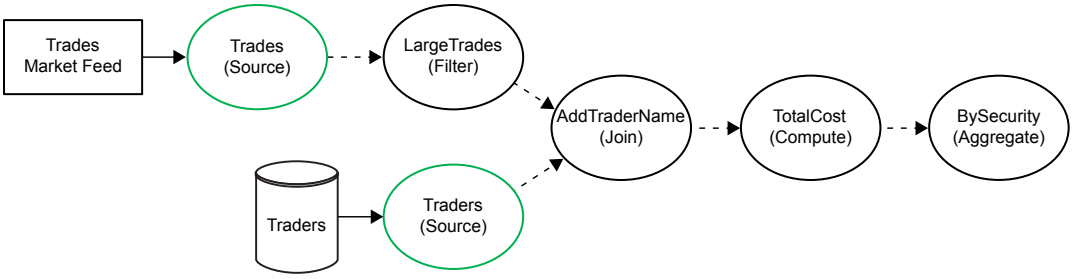
```
-------------------------------------------------------------------------------
filterWindow_01
-------------------------------------------------------------------------------
TID:  0x0005000000000002
depth: 2
     event[0]: <UB,N: 44001,ibm,100.23,3000,100.544000,2010-09-09
     16:09:01,2010-09-09 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>
```

Processing Trades

Overview of the Example

Consider the following continuous query.

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.

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 trades.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 trades.csv -t csv -b 256 -d %d/%b/%Y:%H:%M:%S

Filtering Events

Filtering Events with ESP_OPCODE

The following code demonstrates the use of the ESP_OPCODE reserved word 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 the *SAS Event Stream Processing: User's Guide*.

```
// -*- 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;
}
```

Using the Blue Fusion Standardize Function

Creating a Compute Window that Uses the Blue Fusion Standardize Function

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 the *SAS Event Stream Processing: User's Guide*.

```
// -*- 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;
    }
```

Creating Windows That Use User-Defined Functions

Creating a Compute Window That Uses a UDF

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;
}
```

Creating a Source Window That Uses a Splitter Expression UDF

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;
}
```

Code Examples That Use MAS Modules

XML Example That Uses a MAS Module That Invokes DS2 Code

The following example creates a model with one source window and one procedural window. The procedural window uses DS2 code to calculate a value from the source window. The engine element creates the single engine top level container that sets up fundamental services such as licensing, logging, publish/subscribe. This single engine instance wraps one or more projects. Each project wraps one or more continuous queries and different types of windows.

```
<engine port='55555'>
    <projects>
    <project name='trades_proj' pubsub='auto' threads='4'>
      <description>
        This is to create a project. Project specifies a container
        that holds one or more continuous queries and are backed by a
        thread pool of user defined size. One can specify the pubsub
        port and type, number of threads for the project, index type
        and if using Tag Token data flow model.
      </description>
        <mas-modules>
          <mas-module language="ds2" module="module_1" func-names='compute_volume'>
            <description>
              <![CDATA[This shows a MAS module in DS2]]>
            </description>
            <code>
```

```
        <![CDATA[
        ds2_options sas;
        package module_1/overwrite=yes;
          method compute_volume(int quantity, double price, in_out int volume);
            volume = quantity * price;
          end;
        endpackage;
        ]]>
      </code>
      </mas-module>
    </mas-modules>
<contqueries>
  <contquery name='trades_traders_cq' trace='pw1'>
    <description>
      This specifies the continuous query container that holds
      a collection of windows and enables one to specify the
      connectivity between windows. One can turn on tracing
      for a list of window and specifies the index type for
      windows in the query.
    </description>
    <windows>
      <window-source name='Trades' index='pi_RBTREE'>
        <description>
          This defines a source window. All event streams must
          enter continuous queries by being published or
          injected into a source window.
        </description>
        <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='string'/>
          </fields>
        </schema>
      </window-source>
      <window-procedural name='pw1'>
        <description>
          This defines a procedural window.  The window will pass
          all fields of the event as variables to the DS2 program.
        </description>
        <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='string'/>
            <field name='volume' type='int32' key='true'/>
          </fields>
        </schema>
        <mas-map>
          <window-map module="module_1" revision="0" source="Trades" function="compute_volume"/>
```

```
            </mas-map>
          </window-procedural>
        </windows>
        <edges>
          <description>
            This fully specifies the continuous query with window
            connectivity, which is a directed graph.
          </description>
          <edge source='Trades' target='pw1'/>
        </edges>
      </contquery>
    </contqueries>
  </project>
</projects>
</engine>
```

XML Example That Uses a MAS Module That Invokes Python Code

The following example creates the same windows and performs essentially the same tasks as the previous example. Here though, Python code rather than DS2 code performs the calculation.

```
<project name='trades_proj' pubsub='auto' threads='4'>
      <mas-modules>
      <mas-module language="python" module="module_1" func-names='compute_total'>
      <description>
         <![CDATA[This shows a MAS module in Python]]>
      </description>
        <code>
          <![CDATA[
  def compute_total(quantity, price):
  "Output: total"
  total = quantity * price
  return total
          ]]>
        </code>
      </mas-module>
      </mas-modules>
      <contqueries>
        <contquery name='trades_traders_cq' trace='pw1'>
          <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='string'/>
                </fields>
              </schema>
            </window-source>
            <window-procedural name='pw1'>
              <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='string'/>
                <field name='total' type='double' key='true'/>
              </fields>
            </schema>
            <mas-map>
              <window-map module="module_1" revision="0" source="Trades" function="compute_total"/>
            </mas-map>
          </window-procedural>
        </windows>
        <edges>
          <edge source='Trades' target='pw1'/>
        </edges>
      </contquery>
    </contqueries>
  </project>
```

C++ Example

```
// -*- Mode: C++; indent-tabs-mode: nil; c-basic-offset: 4 -*-

#include <iostream>

// Include class definitions for modeling objects.
//
#include "dfESPevent.h"
#include "dfESPeventblock.h"
#include "dfESPwindow_source.h"
#include "dfESPwindow_procedural.h"
#include "dfESPcontquery.h"
#include "dfESPengine.h"
#include "dfESPproject.h"

using namespace std;

// Declare a callback context data structure to make sure that the
// callback function is thread safe
struct callback_ctx {
    dfESPthreadUtils::mutex *lock;
    dfESPstring windowName;
    };

// Declare a callback function that we register with the procedural
// window, to print out the events that these windows generate.
//

// The callback for the procedural window, note 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 sharing state
// across calls or passing state into calls.
//
void winSubscribe_procedural(dfESPschema *os, dfESPeventblockPtr ob, void *cntx) {
```

```
    dfESPengine::oStream() << endl << "-------------------------------------------------" << endl;
    dfESPengine::oStream() << "proceduralWindow_01" << endl;

    // the dfESPeventblock has a dump() method that prints each
    // event that the event block contains.
    ob->dump(os);
    }

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

    char *ds2_program_01 = (char *)
        "ds2_options sas;"
        "package espmas_test /overwrite=yes;"
        "method compute_total(int ID, int quantity, double price, in_out double total);"
        "total = quantity * price;"
        "end;"
        "endpackage;";

    //
    // -------------- 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 parameter vector as passed into main. currently the dfESP library only
    //        looks for -t <textfile.name> to write its 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.
    //                  -h <http-pubsub port> port for restful pubsub interface. Used for
    //                     accessing esp server from streamviewer.
    // @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, 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,symbol:int32,quantity:int32,price:double"));

// Build the procedural window. We specify the window name, the type
//    of primary index (in this case a red/black tree), and the window
//    schema string.
//
dfESPwindow_procedural *pw_01;
pw_01 = cq_01->newWindow_procedural("proceduralWindow_01",
    dfESPindextypes::pi_RBTREE,
    dfESPstring("ID*:int32,symbol:int32,quantity:int32,price:double,total:double"));

// Publish source code to MAS
// One could have DS2 code in a file and publish sourve code to
// MAS
// pw_01->publishFileToMAS("espmas_test", DS2, "./example.ds2",
//                             "Simple DS2 example.");
//
// One could have DS2 code in a string and publish sourve code to
// MAS
pw_01->publishToMAS("espmas_test", DS2, ds2_program_01,
                            "Simple DS2 example.");

// Generate the context object for the procedural window.
//
dfESPpcontext *ctxt = new dfESPpcontext();

// Register an (input window, handler function) to the context.
//    we only register one pair, as we only have one input window.
ctxt->registerMethod_MAS(sw_01, "espmas_test", "compute_total");

// Attach the contect object to the procedural window.
//
pw_01->registerMethodContext(ctxt);


// Add the subscriber callback to the procedural window.
//    This function get called whenever a window produces output
//    events. The events produced are both passed to the
//    callback function, and also sent further down the directed
//    graph for additional processing. Callback context structure
//    is used to ensure the function thread safe.
//
// In this example, we simple format the events as CSV rows, and
```

```
//    dump them to the screen. This allows us to see what each
//    window is producing at each step of the computation.
//
callback_ctx proc_ctx;
proc_ctx.lock = dfESPthreadUtils::mutex::mutex_create(); // create the lock
pw_01->addSubscriberCallback(winSubscribe_procedural, (void *)&proc_ctx);


// Add the connectivity information to the continuous query. This
//    means sw_o1[slot 0] --> 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 continues query, that is embedded in a
//    project, is running in the background using the defined
//    thread pool. We use the main thread that we are in to inject
//    some data.
//

dfESPengine::oStream() << endl << endl;

// Eeclare some scratch variables to build up and submit the input data.
//
bool eventFailure;
dfESPptrVect<dfESPeventPtr>  trans;
dfESPevent                  *p;
dfESPeventblock             *ib;



//
// ------------ BEGIN - DEFINE AN EVENT AND INJECT to running PROJECT -----------
//

// Build a block of input data (This block only has 1 event in it,
//    but you can have may events in an input block if you like).
//
// the i is insert
// the n is normal
// the 1 is the first field
//    ...
```

```
// the 164.1 is the last field
//
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,10,5,2000,5.00", eventFailure);
if (eventFailure) {
    cerr << "Creating event failed. Aborting..." << endl;
    abort();
    }
trans.push_back(p);
ib =  dfESPeventblock::newEventBlock(&trans, dfESPeventblock::ebt_TRANS);
//
// This clears the vector and frees memory.
//
trans.free();

// Inject the constructed event block into the graph. Note this
//    call is asynchronous with respect to processing. It deposits
//    the input block to the queue of the source window, and the
//    thread pool assigned to the project takes over the execution
//    from there.
//
// When we return from this call all we know is
//    that the input event has been queued up for processing.
//
project_01->injectData(cq_01, sw_01, ib);

//
// ------------ END - DEFINE AN EVENT AND INJECT to running PROJECT -----------
//


// We repeat the above event definition and injection into
//     the project with a much less verbose amount of comments
//     below.
//
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,20,50,1000,10.00", 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);


// We repeat the above event definition and injection into
//     the project with a much less verbose amount of comments
//     below.
//
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,30,55,4750,3.49", 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);


// We repeat the above event definition and injection into
//     the project with a much less verbose amount of comments
//     below.
//
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,40,100,3000,15.00", 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);



// We repeat the above event definition and injection into
//     the project with a much less verbose amount of comments
//     below.
//
p = new dfESPevent(sw_01->getSchema(),(char *)"i,n,50,150,3000,20.25", 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();  // quiesce the graph of events

dfESPengine::oStream() << endl << endl;

// Cleanup & shutdown.
//
myEngine->shutdown();

return 0;
}
```

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. Copyright © 2016, SAS Institute Inc., Cary, NC, USA. All Rights Reserved. September 2016 4.2-P1:espex