SAS® Event Stream Processing 5.1: Examples

Overview

SAS Event Stream Processing code examples are available in **`$DFESP_HOME/examples`**. Examples that are written in various programming languages are available in subdirectories of this directory.

Programming Language	Subdirectory
C++	cxx
XML	xml
Java	java
Python	python

Peruse the file readme.examples.txt for more information.

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 *schema* that you specify. 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:

2

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

The Filter window inherits this schema from the Source window.

In this case, 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 “How to Build and Run the Source Code”.

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
20	MS	3000	26.99	20	MS	3000	26.99
30	ACL	2000	2.11	30	ACL	2000	2.11

Code for the Example

C++ code that implements this example is available in **`$DFESP_HOME/examples/cxx/filter_exp`**. Edit the associated Makefile to remove the comments for architecture-specific build variables.

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/examples/cxx/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:

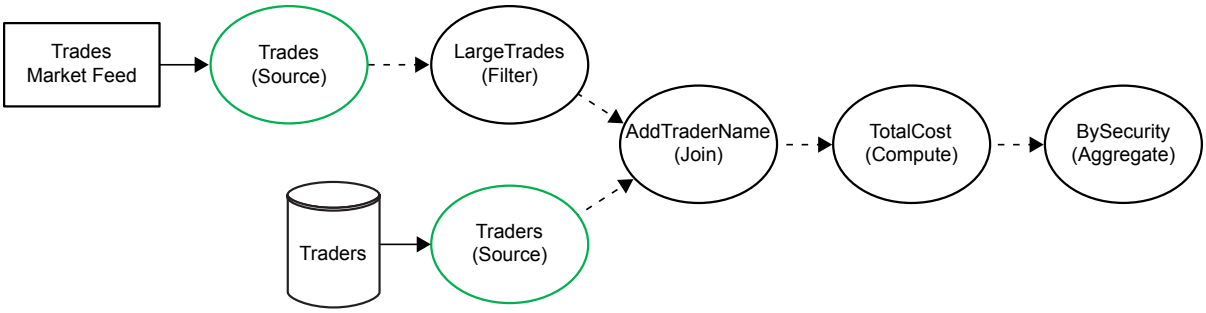
```
# -- GCC on Linux
#    Uncomment the next three lines to use these settings.
# CXX=g++
# CXXFLAGS=-g -m64
# LDFLAGS=-L$$DFESP_HOME/lib
```

Processing Trades

The Continuous Query and the Associated XML Code

Consider the following continuous query.

Continuous Query Diagram



Here is how you would implement that query in the SAS Event Stream Processing XML modeling language. You can find this code in **$DFESP_HOME/examples/trades_xml**.

```
<engine name='trades' port='55555' dateformat='%d/%b/%Y:%H:%M:%S'> 1
    <projects>
        <project name='trades_proj' pubsub='auto' threads='4'> 2
            <contqueries>
                <contquery name='trades_cq'> 3
                    <windows>
```

1 An engine named trades on a publish/subscribe port of 5555 is established.

2 A project named trades_proj with a publish/subscribe mode of auto is established. Four threads are used from the available thread pool.

3 A continuous query named trades_cq is established.

```
<window-source name='Trades' index='pi_HASH'> 1
       <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'> 2
      <schema>
          <fields>
              <field name='ID' type='int64' key='true'/>
              <field name='name' type='string'/>
          </fields>
      </schema>
</window-source>
```

1 A Source window named Trades is established with a pi_HASH index type. This window streams data about securities transactions from a trades market feed.

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

A Filter window named LargeTrades is established to receive events from the Trades window. It filters out any event that involve fewer than 100 shares.

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

A Join window named AddTraderName performs a join operation with values from the two Source windows. It matches filtered transactions with their associated traders

```
<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>
```

A Compute window named TotalCost uses data from the Join window to calculate the cost of the transaction.

```
<window-compute name='TotalCost'>
       <schema>
           <fields>
               <field name='tradeID' type='string' key='true'/>
               <field name='security' type='string'/> 1
               <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>
       <output> 2
          <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>
```

1 This field and the following fields are the non-key fields.

2 This is how the non-key fields are computed.

An Aggregate window is established named BySecurity. The ESP_aSum function is used to sum total quantity and cost.

```
<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 connect the windows.

```
<edges>
           <edge source='LargeTrades' target='AddTraderName'/> 1
           <edge source='Traders' target='AddTraderName'/>
           <edge source='Trades' target='LargeTrades'/> 2
           <edge source='AddTraderName' target='TotalCost'/>
           <edge source='TotalCost' target='BySecurity'/>
       </edges>
    </contquery>
  </contqueries>
</project>
</projects>
</engine>
```

1 The Filter window and the Traders Source window flow events to the Join window.

2 The Trades window flows events to the Filter window. The Join window flows events to the Compute window, which flows events to the Aggregate window

Running the XML Code

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

1 Execute the ESP server with the model:

$DFESP_HOME/bin/dfesp_xml_server -model file://*full_path_to_xmlfile* -http 61001.

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 Streamviewer to subscribe to the final BySecurity window to see the computed data as the trades data flows through the model.

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

To run the example code on Windows systems, change **$DFESP_HOME/bin** to **%DFESP_HOME%\bin**.