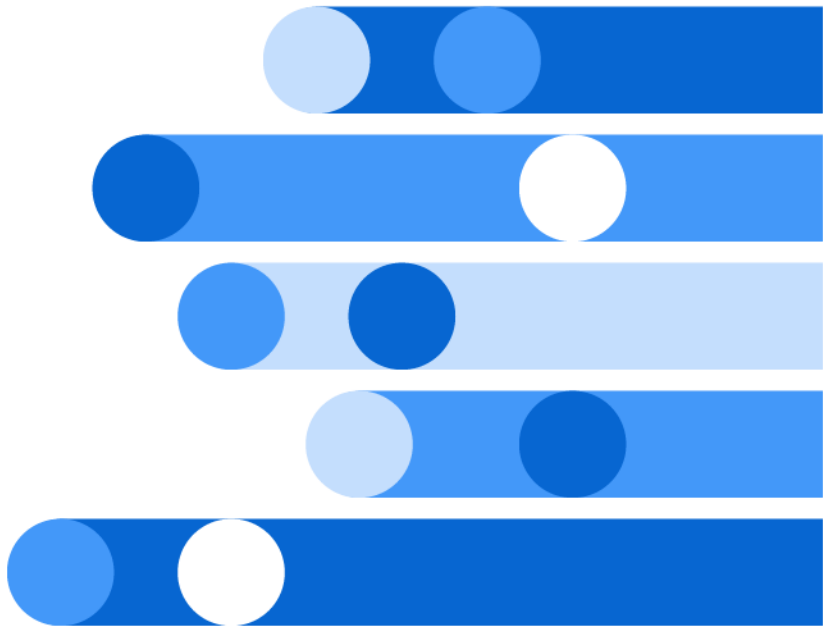
SAS® Event Stream Processing: Using Source and Derived Windows

2026.08*



* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.

SAS® Documentation
August 18, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2026. *SAS® Event Stream Processing: Using Source and Derived Windows*. Cary, NC: SAS Institute Inc.

SAS® Event Stream Processing: Using Source and Derived Windows

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

August 2026

v_035-P1:espcreatewindows

Contents

PART 1

Source and Derived Windows

1

Overview

Window Types

An event stream processing project specifies how input event streams are transformed and analyzed into meaningful resultant event streams. Each project contains one or more continuous queries.

A continuous query contains one or more Source windows and one or more *derived windows*. All event streams must enter continuous queries by being published or injected into a Source window. Event streams cannot be published or injected into any other window type.

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

SAS Event Stream Processing supports a variety of derived window types, each having a specialized purpose.

Table 1.1 *Derived Window Types*

Window Type	Description
Compute window	Enables a one-to-one transformation of input events into output events through the computational manipulation of the input event stream fields.
Aggregate window	Similar to a Compute window in that non-key fields are computed. An Aggregate window uses the key field or fields for the group-by condition. All unique key field combinations form their own group within the Aggregate window. All events with the same key combination are part of the same group.

Window Type	Description
Calculate window	Transform data events using a variety of analytical algorithms.
Copy window	Makes a copy of the parent window. Making a copy can be useful to set new event state retention policies. Retention policies can be set only in source and Copy windows. You can set event state retention for a Copy window only when the window is not specified to be Insert-only and when the window index is not set to `pi_EMPTY`. All subsequent sibling windows are affected by retention management. Events are deleted when they exceed the windows retention policy.
Counter window	Enables you to see how many events are streaming through your model and the rate at which they are being processed.
Filter window	Uses a registered Boolean filter function or expression. This function or expression determines what input events are allowed into the Filter window.
Functional window	Enables you to use different types of functions to manipulate or transform the data in events. Fields in a Functional window can be hierarchical, which can be useful for applications such as web analytics.
Geofence window	Enables you to create a window to determine whether the location of an event stream is inside or near an area of interest.
Join window	Takes two input windows and a join type. Supports equijoins that are one to many, many to one, or many to many. Both inner and outer joins are supported.
Lua window	Enables you to generate SAS Event Stream Processing events with Lua programs.
Model Reader window	Read models brought into SAS Event Stream Processing as analytic store files or as a combination of non-binary files.
Model Supervisor window	Manage models received from Model Reader windows.
Notification window	Enables you to send notifications through email, text, or multimedia message. You can create any number of delivery channels to send the notifications. A Notification window uses the same underlying language and functions as the Functional window.

Window Type	Description
Object Tracking window	Enables you to perform multi-object tracking (MOT) in real time.
Pattern window	Enables the detection of events of interest (EOI). A pattern defined in this window type is an expression that logically connects declared events of interest. To define a pattern window, you need to define events of interests and then connect these events of interest using operators. The supported operators are "AND", "OR", "FBY", "NOT", "NOTOCCUR", and "IS". The operators can accept optional temporal conditions.
Procedural window	Enables the specification of an arbitrary number of input windows and input-handler functions for each input window (that is, event stream).
Python window	Enables you to generate SAS Event Stream Processing events with Python programs.
Remove State window	Facilitates the transition of a stateful part of a model to a stateless part of a model.
Score window	Score events with online analytical algorithms that are packaged with SAS Event Stream Processing or with algorithms in offline models.
StateDB Reader	Query or fetch data from a defined external database based on the incoming event and specified criteria. These criteria are based in part on the indexes you create in the StateDB Writer window.
StateDB Writer	Write an incoming event to the defined external database, create required indexes for the event, and pass the input event unchanged to the next window of a project.
Train window	Refine the algorithm parameters of online models based on streaming event data.
Transpose window	Enables you to interchange an event's rows as columns, or columns as rows.
Union window	Combines multiple event streams with the same schema into a single stream, similar to an SQL union operation.

For more information about Calculate windows, Model Reader windows, Model Supervisor windows, Score windows, and Train windows, see *SAS Event Stream Processing: Using Streaming Analytics*.

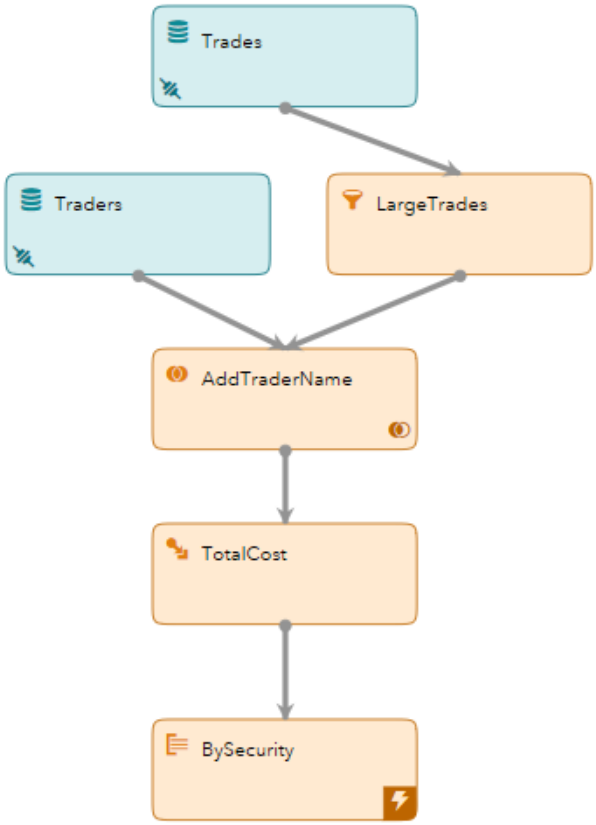
Creating a Continuous Query with Windows and Edges

Suppose that you wanted to create a continuous query to process stock trades. For that query, you want to do the following:

- publish two events streams into the query: one for trades and another for the corresponding traders
- filter out trades of less than 100 shares
- compute the total trade cost after all shares are transacted
- write a file grouped by security that shows total cost

You can create the project in SAS Event Stream Processing Studio.

Figure 1.1 *Continuous Query Diagram*



Here is the corresponding SAS Event Stream Processing XML modeling language. You can download this code from the support web site. Navigate to / **`FurtherExamples/trades`** to obtain the XML file and supporting CSV files.

```
<project name='trades_project' pubsub='auto' threads='4'> <!--1-->
            <contqueries>
                <contquery name='trades_cq'> <!--2-->
                    <windows>
```

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

2 A continuous query named trades_cq is established.

```
<window-source name='Trades' insert-only='true' index='pi_EMPTY'>
<!--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>
       <connectors>
            <connector class="fs" name="publisher">  <!--2-->
              <properties>
                <property name="type">pub</property>
                <property name="fstype">csv</property>
                <property name="fsname">trades.csv</property>
                <property name="transactional">true</property>
                <property name="blocksize">1</property>
                <property name="dateformat">%d/%b/%Y:%H:%M:%S</
property>
              </properties>
            </connector>
        </connectors>
</window-source>

<window-source name='Traders' insert-only='true' index="pi_EMPTY">
<!--3-->
      <schema>
          <fields>
              <field name='ID' type='int64' key='true'/>
              <field name='name' type='string'/>
          </fields>
      </schema>
     <connectors>
            <connector class="fs" name="publisher">  <!--4-->
              <properties>
                <property name="type">pub</property>
                <property name="fstype">csv</property>
                <property name="fsname">traders.csv</property>
                <property name="transactional">true</property>
                <property name="blocksize">1</property>
              </properties>
            </connector>
        </connectors>
</window-source>
```

1. A Source window named Trades is established with a pi_EMPTY index type. This window streams data about securities transactions from a CSV file.
2. A file and socket connector is established to publish events into the Trades window from a file in the current working directory named trades.csv.
3. 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.
4. A file and socket connector is established to publish events into the Traders window from a file in the current working directory named traders.csv.

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' index="pi_EMPTY">
       <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' index="pi_EMPTY">
       <join type="leftouter" no-regenerate="true">
           <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' index='pi_EMPTY'>
       <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>
          <field-expr>security</field-expr>
          <field-expr>quantity</field-expr>
```

```
            <field-expr>price</field-expr>
            <field-expr>price*quantity</field-expr> <!--2-->
            <field-expr>traderID</field-expr>
            <field-expr>time</field-expr>
            <field-expr>name</field-expr>
        </output>
</window-compute>
```

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

2 This is how totalCost is computed.

An Aggregate window is established named BySecurity. The ESP_aSum function is used to sum total quantity and cost. A subscriber connector publishes the output to a CSV file named result.out.

```
 <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>
        <connectors>
            <connector class="fs" name="sub">
              <properties>
                <property name="type">sub</property>
                <property name="fstype">csv</property>
                <property name="header">true</property>
                <property name="fsname">result.csv</property>
                <property name="snapshot">true</property>
              </properties>
            </connector>
          </connectors>
</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>
```

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

The output CSV file shows two retained events. The first indicates that, after all trades were transacted, 1000 shares of IBM were sold at a total cost of $100,100.00. The second indicates that 750 shares of SAP were sold at a total cost of $25,650.00.

Figure 1.2 *Output from the Aggregate Window*

security	quantityTotal	costTotal		
I	N	ibm	1000	100100
I	N	sap	750	25650
UB	N	ibm	2000	200300
D	N	ibm	1000	100100
UB	N	ibm	3000	300600
D	N	ibm	2000	200300
UB	N	ibm	4000	401000
D	N	ibm	3000	300600
UB	N	sap	1750	59950
D	N	sap	750	25650
UB	N	ibm	5000	501300
D	N	ibm	4000	401000
UB	N	sap	2750	91950
D	N	sap	1750	59950
UB	N	ibm	6000	601300
D	N	ibm	5000	501300

2

Input Streams

Overview to Source Windows

At least one Source window is required for each continuous query. All event streams enter continuous queries by being published or injected into a Source window. Event streams cannot be published or injected into any other window type.

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

Source windows accept streaming data or raw data files through in-process connectors or executable adapters. Source windows can also accept data from the publish/subscribe API, HTTP clients, or by injection from the C++ Modeling API.

Ordinarily, a Source window queues incoming events until downstream derived windows are free to process them. Because event blocks are never dropped, there might be considerable backup in the Source window whenever downstream processing is significant or resources are limited. You can use the `queue-height` parameter to set the size of the input queue. At times it can be useful to not queue incoming events at all, processing only the latest event. Use the `shed-input` parameter to force the Source window to process only the latest available event block and avoid backup.

For information about the XML elements associated with Source windows, see "window-source" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*

Defining Event Index Types

Source windows can have a primary index and a retention index. There are four fully stateful index types and one stateless index type. The primary index type of a source window affects the performance of the rest of the project.

For example, when a Source window has index type `pi_RBTREE`, the window is stateful and retains all incoming events. Retaining events at the Source window without a retention policy uses substantially more memory as data is read into the continuous query than when the Source window is stateless.

For more information about index types, see "Understanding Primary and Specialized Indexes".

Retention Policies in Source Windows

Retention policies limit the flow of incoming data by time or event count, and thus can control the total number of events that stream through the model. Events are deleted automatically when they exceed the window's retention policy.

You can define a retention policy in stateful Source and Copy windows. To use retention policies, the window cannot be specified as Insert-only and the index type cannot be specified as `pi_EMPTY`. Usually, you want to follow any insert-only Source window with a Copy window with a retention policy.

For more information about retention policies, see "Understanding Retention".

Propagation of Insert-Only Processing

You can explicitly set a Source window to accept only Insert events. This can optimize event stream processing. Insert-only processing propagates to all derived windows unless one of the following conditions is met:

- A window is determined not to produce Insert-only data (for example, a window with retention).

- The *ESP server* cannot determine whether the derived window produces only Inserts:
  - When a Procedural or Calculate window has set `algorithm='MAS'`, it runs an arbitrary block of user-written code that could result in a non-Insert event. If you are certain that your code produces only Inserts, explicitly add the attribute `produces-only-inserts='true'` to the XML specification of the window.
  - When a Functional window uses a function that changes the opcode of a produced event. You can override this with the attribute `produces-only-inserts='true | false'`.
  - The Aggregate window produces numerous Updates. It always sets `produces-only-inserts='false'`.

Automatically Generating Key Values

Source windows can automatically generate identification keys for incoming events. To automatically generate key values, the Source window must be insert-only and have only one key with type INT64 or STRING. The `autogen-key` attribute of the `window-source` element must be set to **`true`**.

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

The Publisher Connector

The publisher connector is unique to the Source window. It determines the following:

- the path of the data source
- the data type of the events received from the data source
- other important properties that are related to translating incoming data into events that are used throughout the project

The Source window's publisher connector determines how the data is read and in what format events are pushed to derived windows. In XML code, the required and optional properties of the connectors are defined in the connector element.

The fields defined by the schema of the Source window determine the structure of the data read into the project and used by derived windows.

Enabling Metering Windows

A metering window tracks the number of events processed (and related timestamps) by all of the Source windows in a model. To enable this functionality, you must define the field `enableMetaProject` with the value **`true`**. This sets up the metering window in a continuous query named _meta_, which is contained in a project named _meta_.

Source windows inject events into the metering window at a default interval of five seconds. Subscribing applications can subscribe to the metering window as they do for any other window.

The metering window is itself a special Source window named _eventmetering_. It has the following schema:

```
"project*:string,query*:string,window*:string,currenttime:stamp,lasttime:stamp,numevents:int64"
```

You can append string fields to the end of the metering window schema when you start a metering server. Subsequently, all metering events generated by the metering window contain those fields. You can define or redefine the values of those fields at any time. If not defined at start-up, the values are null until you define them. You can also reconfigure the default metering interval of five seconds.

When you run the metering server and its `meteringhost` and `meteringport` parameters are set, each update to a metering window is forwarded to the metering server for aggregation.

Using Singletons

A Source window that is defined within a continuous query but not included in an edge element is called a singleton. Singletons can be useful to validate a connection between the outside world and the *ESP server* and validating input formats when designing a project.

For example, suppose that the primary input into your project is a message bus that streams events that are in JSON format. You have an idea of the format but are uncertain about the actual content. In this case, you can design a singleton with a publisher connector to test the connection, inspect the JSON passing parameters, and inspect the incoming data. You could use SAS Event Stream Processing Studio in test mode to inspect data flows.

Afterward, you might want to collect a snapshot of events to a static data set for closer inspection. You could add a publisher connector to the singleton to store data to a specified file.

You specify whether or not to include singletons in a continuous query using the `include-singletons` attribute of the `contquery` element. For more information, see "contquery" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

3

Transformations

Using Filter Windows

Overview

Filter windows can use Lua code, Python code, expressions in the Expression Engine Language (EEL), user-defined functions (global functions), and registered plug-in functions to determine whether input events are permitted to pass through a project. These functions and expressions are called *filter conditions*. When a filter condition is true, an input event passes through. When a filter condition is false, an event is blocked.

When the *ESP server* detects a `code` element in the Filter window, the server evaluates the code for the filter condition instead of using the expression language.

For information about additional Lua functionality that is available when you use Lua in a Filter window, see Chapter 8, “Common Lua Functionality”.

For information about additional Python functionality that is available when you use Python in a Filter window, see Chapter 9, “Common Python Functionality”

For more information about creating filter conditions that use the Expression Engine Language, see“Overview to Expressions”.

“window-filter” in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*For information about the XML elements that are associated with Filter windows, see and “XML Language Elements Relevant to Compute and Filter Windows” in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Using Lua in a Filter Window

In a Lua code block, you must specify a filter function that returns a Boolean value. The expected name of the filter function is `filter`. You can use the `func` attribute of the `window-filter` element to specify an alternative name for the filter function. The *ESP server* calls the filter function for each input event and uses the return code to determine whether to generate an output event.

For example, the following XML code permits all events to pass through whenever the value of the **`count`** input field is greater than or equal to 100:

```
<window-filter name="filter_w">
  <code><![CDATA[
    function filter(event)
        return event.count >= 100
```

```
        end
      ]]></code>
    </window-filter>
```

You can include a use element in the Filter window to restrict what fields from the input event are processed by the Lua filter function. Processing efficiency can be improved by passing fewer input fields to Lua.

For example, a Trades project could include hundreds of input fields, but you might want to filter events based on the value of a single input field. The following example restricts the filter condition to the value of a single input field, `quant`.

```
<window-filter name="largeTrades" func="bob">
    <use>quant</use>
    <code><![CDATA[
        function bob(data)
            return data.quant >= 1000
        end
    ]]></code>
</window-filter>
```

If the `use` element is missing, then all input fields are passed to the Lua filter function.

If you set the element to no fields (`<use/>`), then no fields are passed to the Lua filter function. In this case, you must access the fields on the *ESP server* side by using function calls. For example, if you had JSON data streaming through a window that you wanted to process in Lua, you could specify `<use/>`, and then use `esp_parseJsonFrom('field')` to fetch the data.

Note: Although the `use` element is optional, as a best practice, use it to pass the minimum number of fields from the *ESP server* into the Lua filter function.

The parameters of the `filter` function are as follows:

Table 3.1 *Parameters of the Filter Function*

Parameter	Description
object	Specifies a Lua object that contains all the input event fields. If you specify a `use` element, then the specified event fields are included. This parameter is required.
context	Specifies context data for the current event. The valid keys are as follows: ■ **`window`** - The name of the current window. ■ **`input`** - The name of the input window for this event. This parameter is optional.

If you need to perform initialization in your Lua code, specify an `init` attribute for the Filter window. The value of this attribute must be a valid Lua function. No parameters are passed to the `init` function.

If you need to perform cleanup in your Lua code, specify a `destroy` attribute for the Filter window. The value of this attribute must be a valid Lua function. This function is called when the Filter window is destroyed.

If you want to bring in internal Lua modules, specify them in a `require` attribute for the Filter window. Specify a comma-separated list of internal Lua modules to include.

To perform intermittent processing, the Lua-based Filter window supports a `heartbeat` attribute. The value of this attribute must be a valid Lua function. No parameters are passed to the heartbeat function.

Using Python in a Filter Window

Note: When you use a Python-based Filter window, your code must be compatible with Python 3.11.

In a Python code block, you must specify a filter function that returns a Boolean value. The expected name of the filter function is `filter`. You can use the `func` attribute of the `window-filter` element to specify an alternative name for the filter function. The ESP server calls this function for each input event and uses the return code to determine whether to generate an output event.

For example, the following XML code permits all events to pass through whenever the value of the **`count`** input field is greater than or equal to 100:

```
<window-filter name="filter_w" func="bob">
  <code><![CDATA[
def bob(event, context):
    return event["the_count"] >= 100
  ]]></code>
</window-filter>
```

You can include a use element in the Filter window to restrict what fields from the input event are processed by the Python filter function.

For example, a Trades project could include hundreds of input fields, but you might want to filter events based on the value of a single input field. The following example restricts the filter condition to the value of a single input field, `quant`.

```
<window-filter name="largeTrades">
    <use>quant</use>
    <code><![CDATA[
def filter(data, context):
    return data["quant"] >= 1000
    ]]></code>
</window-filter>
```

If the `use` element is missing, then all input fields are passed to the Python filter function.

The parameters of the `filter` function are as follows:

Table 3.2 *Parameters of the Filter Function*

Parameter	Description
event	Specifies the input event. If you specify a `use` element, then the specified event fields are included. This parameter is required.
context	Includes variables, data, and resources that are available for the filter function to use. In a Filter window, context can contain the following information: ■ **`window`** - The ID of the containing Python window. ■ **`input`** - The name of the input window for this event.

If you need to perform initialization in your Python code, specify an `init` attribute for the Filter window. The value of this attribute must be a valid Python function. No parameters are passed to the `init` function.

If you need to perform cleanup in your Python code, specify a `destroy` attribute for the Filter window. The value of this attribute must be a valid Python function. This function is called when the Filter window is destroyed.

To perform intermittent processing, the Python-based Filter window supports a `heartbeat` attribute. The value of this attribute must be a valid Python function. No parameters are passed to the heartbeat function.

Using Aggregate Windows

Overview to Aggregate Windows

Aggregate windows are similar to Compute windows in that non-key fields are computed. However, you must specify key fields of Aggregate windows; they are not inherited from the input window. Those key fields must correspond to existing fields in the input event. Incoming events are placed into aggregate groups. Each event in an aggregate group has identical values for the specified key fields.

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

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

Now suppose that you specify the schema for an Aggregate window as follows:

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

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

These groups are internally maintained in the `dfESPwindow_aggregate` class as `dfESPgroupstate` objects. Each group is collapsed every time that a new event is added or removed from a group by running the specified aggregate functions or expressions on all non-key fields. The Aggregate window produces one aggregated event per group.

For information about the XML elements associated with Aggregate windows, see "window-aggregate" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Flow of Operations

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

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

Using Aggregate Functions

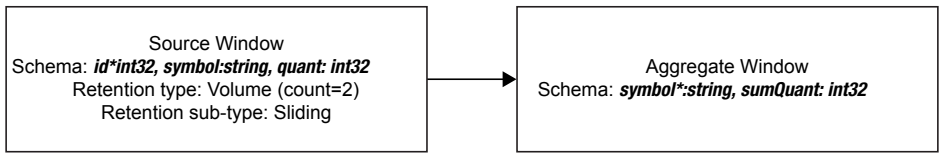
Overview to Using Aggregate Functions

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

The functions that are specified for non-key fields of the Aggregate window are special functions. They operate on groups of event values and collapse them into a single scalar value.

In the following diagram, the key of the Aggregate window is **symbol**. The Aggregate window has only one non-key field, **sumQuant**. This field holds the sum of the field **quant** that arrives from the Source window.

Figure 3.1 *Example of Aggregation*



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

You can use aggregate functions on arrays. For example, suppose that you have the following input schema:

```
name*: string, value: array
```

Suppose that the Source window feeds the following five events into an Aggregate window:

Table 3.3 *Events Streaming through a Source Window*

Event	Name (Key)	Value
e1	A	[1.0, 2.0]
e2	A	[0.0, 7.0]
e3	B	[5.0, 2.0]
e4	A	[3.0, 1.0]
e5	B	[3.0, 4.0]

Now suppose that you apply `ESP_aMax(value)` in an Aggregate window that receives these events from the Source window. First, the Aggregate window groups data by the key field `name`.

Table 3.4 *Grouping Events by the Key Field*

Key	Values
A	`[1,0, 2.0]` `[0.0, 7.0]`

Key	Values
	[3.0, 1.0]
B	[5.0, 2.0] [3.0, 4.0]

The ESP_aMax(value) function in the Aggregate window works position-wise on the arrays. The maximum value of the first element of the array values for A is 3.0, and the maximum value of the second element is 7.0. Thus, for A, the function produces this result:

```
[3.0, 7.0]
```

For B, the function produces this result:

```
[5.0, 4.0]
```

Aggregate functions return a result even when arrays are not symmetrical. For example:

```
ESP_aSum([1,2,3],[1,2],[3])
```

Returns [5]: 1+1+3=5. The length of the result in cases such as these is the length of the shortest array.

Aggregate functions that count fields, such as aCount,aCountNonNull, and aCountNull, or functions that choose a field from a group, such as aFirst, aLast, and aLastNonNull, operate on the entire array. They do not inspect the elements of the array. Suppose that you stream the following data into an Aggregate window:

```
[1;2;3]
(NULL)
[3;4;5]
(NULL)
```

Table 3.5 *Aggregate Functions Applied to Array Data*

Function	Returns
aCount	4
aCountNonNull	2
aCountNull	2
aFirst	[1;2;3]
aLast	(NULL)
aLastNonNull	[3;4;5]

For the count functions, the return type should be int64. For the first and last functions, the type should be the same array type as the input data.

You can write your own aggregate functions. For more information, see "Writing Aggregate Functions to Embed in Applications".

Aggregate Functions for Aggregate Window Field Calculation Expressions

The following aggregate functions are available for Aggregate window field calculation expressions. Additive functions can be calculated from retained state and the values determined from the incoming event. They do not need to maintain a group state. If these are the only functions that you use in an Aggregate window, special optimizations of an order of magnitude can occur in window processing.

Some aggregate functions are additive regardless of the opcode of the incoming event. Others are additive only when they get Inserts. Some functions can handle array input.

Table 3.6 *Aggregate Functions*

Aggregate Function	Additive	Additive for Inserts	Arrays	Returns
`ESP_aAve(`*`fieldname`*`)`	True	True	Position-wise	Average of the group.
`ESP_aAveTimed(`*`fieldname`*`, `*`N`*`)`	True	True	Position-wise	Average over the events in a group. When a new event arrives *N* or more seconds after the first event, clear the average. This enables timed averages that clear without having to introduce event retention.
`ESP_aCat(`*`fieldname`*`, `*`stringConstant`*`)`	False	True	False	A concatenation of *fieldname* values for each event in an aggregation group using the specified *stringConstant* as the separator. For example, `ESP_aCat(ID, "|")` returns a "|" separated list of all the IDs that make up the aggregation group.

Aggregate Function	Additive	Additive for Inserts	Arrays	Returns
`ESP_aCount()`	True	True	False	Number of events in the group.
`ESP_aCountDistinct(`*`fieldname`*`)`	True	True	False	Number of distinct non-null values in the column specified by *fieldname* within a group.
`ESP_aCountNonNull(`*`fieldname`*`)`	False	True	Counts when the array is not NULL.	Number of events with non-null values in the column for the specified *fieldname* within the group.
`ESP_aCountNull(`*`fieldname`*`)`	False	True	Counts when the array is NULL	Number of events with null values in the column for the specified *fieldname* within the group.
`ESP_aCountTimed(`*`N`*`)`	True	True	False	Count of the number of events in a group. When a new event arrives *N* or more seconds after the first event, clear the count. This enables timed counts that clear without having to introduce event retention.
`ESP_aCountOpcodes(`*`opcode`*`)`	True	True	False	Count of the number of events matching *opcode* for group. Specify the *opcode* with an integer value. ■ NOOP = 0 ■ INSERT = 1 ■ UPDATE = 2 ■ DELETE = 3

Aggregate Function	Additive	Additive for Inserts	Arrays	Returns
				■ UPSERT = 4 ■ SAFEDELETE = 5 ■ UPDATEBLOCK = 6 For example, specify the following to count events that match DELETE. `ESP_aCountOpcodes(3)`
`ESP_aFirst(fieldname)`	False	True	Position-wise	First event added to the group.
`ESP_aGUID()`	True	True	False	A unique identifier.
`ESP_aLag(fieldname,lag_value)`	False	True	Position-wise	The specified *lag_value* represents an index into a group of aggregated event values. An index of 0 represents the current event that affected the group. An index of 1 represents the immediately previous event that affected the group, and so on. Thus, the following holds: ■ `ESP_aLag(fieldname,0) == ESPaLast(fieldname)` returns the value of *fieldname* for the current event that affected the group. ■ `ESP_aLag(fieldname,1)` returns the value of *fieldname* for the immediately previous event that affected the group.
`ESP_aLast(fieldname)`	False	True	Position-wise	Field from the last record that affected the group.

Aggregate Function	Additive	Additive for Inserts	Arrays	Returns
`ESP_aLastNonNull(`*`fieldname`*`)`	False	True	Position-wise	Field from the last record with non-null value that affected the group.
`ESP_aLastOpcode()`	True	True	False	The opcode of the last record to affect the group.
`ESP_aMax(`*`fieldname`*`)`	False	True	Position-wise	Maximum of the group.
`ESP_aMaxTimed(`*`fieldname, N`*`)`	True	True	Position-wise	Maximum value of the group. When a new event arrives *N* or more seconds after the first event, clear the maximum value. This enables timed maximums that clear without having to introduce event retention.
`ESP_aMin(`*`fieldname`*`)`	False	True	Position-wise	Minimum value of the group.
`ESP_aMinTimed(`*`fieldname, N`*`)`	True	True	Position-wise	Minimum value of the group. When a new event arrives *N* or more seconds after the first event, clear the minimum value. This enables timed minimums that clear without having to introduce event retention.
`ESP_aMode(`*`fieldname`*`)`	True	True	False	Mode, or most popular of the group.
`ESP_aStd(`*`fieldname`*`)`	True	True	Position-wise	Standard deviation of the group.
`ESP_aSum(`*`fieldname`*`)`	True	True	Position-wise	Sum of the group.

Aggregate Function	Additive	Additive for Inserts	Arrays	Returns
`ESP_aSumTimed(`*`fieldname, N`*`)`	True	True	Position-wise	Sum of the group. When a new event arrives *N* or more seconds after the first event, clear the sum. This enables timed sums that clear without having to introduce event retention.
`ESP_aWAve(`*`weight_fieldname, payload_fieldname`*`)`	True	True	Position-wise	Weighted group average.
`ESP_aWAveTimed(`*`weight_fieldname, payload_fieldname, N`*`)`	True	True	Position-wise	Weighted group average. When a new event arrives *N* or more seconds after the first event, clear the average. This enables timed weighted averages that clear without having to introduce event retention.

All aggregate functions are additive when you feed them Insert events. Thus, consider the following points with regard to model performance:

- When an insert-only Aggregate window has a key with an unbounded number of elements, the window experiences unlimited memory growth. This cannot be automatically detected by an *ESP server*, because you cannot predict the values passed by incoming events. In this case a WARNING is issued.
- For Aggregation windows that are not Insert-only, the window can be one of the following:
  - Additive, that is, one that uses only the aggregation functions that are marked as always additive. If the cardinality of the keys is unbounded, then the window experiences unlimited memory growth. Because it is not receiving Inserts, no WARNING is issued. Here, you should put a Copy window with retention in front of the Aggregate window in order to control the total number of events in the Aggregate window.
  - Nonadditive, that is, one that uses nonadditive aggregation functions such as ESP_aMax or ESP_aMin. In this case memory growth is primarily driven by the maximum number of events that are active in the Aggregate window at any time. For this case, the Aggregate window maintains a copy of each event that streams in. It also keeps track of what group the event belongs to. Here, it is recommended that you precede the Aggregate window with a Copy window with retention. That ensures that there is always a finite

number of events in the Aggregate window. The number of events would be determined by retention policy.

Event history affects performance:

- Some aggregate functions (for example, `ESP_aMin` and `ESP_aMax`) require source events to be stored (history) in order to process Updates and Deletes.
- All aggregate functions can handle Insert-only event streams, like sensor readings, without history.
- For event streams with Updates and Deletes, use only aggregate functions that do not require history (for example, `ESP_aSum`, `ESP_aAve`) for best performance.

Source event history, when applicable, is kept in an internal index.

You can easily use aggregate functions for non-key field calculation expressions. For example:

```
<window-aggregate name='brokerAlertsAggr' index='pi_HASH'>
   <schema-
string>brokerName*:string,frontRunningBuy:int32,frontRunningSell:int32,

openMarking:int32,closeMarking:int32,restrictedTrades:int32,total:int64
   </schema-string>
   <output>
      <field-expr>ESP_aSum(frontRunningBuy)</field-expr>
      <field-expr>ESP_aSum(frontRunningSell)</field-expr>
      <field-expr>ESP_aSum(openMarking)</field-expr>
      <field-expr>ESP_aSum(closeMarking)</field-expr>
      <field-expr>ESP_aSum(restrictedTrades)</field-expr>
      <field-expr>ESP_aSum(total)</field-expr>
   </output>
</window-aggregate>
```

Note: In `ESP_aSum`, `ESP_aMax`, `ESP_aMin`, `ESP_aAve`, `ESP_aStd`, and `ESP_aWAve`, null values in a field are ignored. Therefore, they do not contribute to the computation.

Using an Aggregate Function to Add Statistics to an Incoming Event

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

Suppose that you are processing stock transactions. The Source window that processes incoming events that contains several fields.

```
<schema>
    <fields>
      <field name="ID" type="int32" key="true"/>
```

```
      <field name="symbol" type="string"/>
      <field name="currency" type="int32"/>
      <field name="udate" type="int64"/>
      <field name="msecs" type="int32"/>
      <field name="price" type="double"/>
      <field name="quant" type="int32"/>
      <field name="venue" type="int32"/>
      <field name="broker" type="int32"/>
      <field name="buyer" type="int32"/>
      <field name="seller" type="int32"/>
      <field name="buysellflg" type="int32"/>
    </fields>
  </schema>
```

You want to restrict the incoming event schema for the Aggregate window to three variables: `symbol`, `price`, and `quant`. To those variables, you want to add two aggregate statistics: `priceAVE` and `quantMAX`.

```
<schema>
    <fields>
      <field name="symbol" type="string" key="true"/>
      <field name="price" type="double"/>
      <field name="quant" type="int32"/>
      <field name="priceAVE" type="double"/>
      <field name="quantMAX" type="int32"/>
    </fields>
  </schema>
```

Note: The groupBy is the key of the aggregation, which in this case is `symbol`.

Use `field-expr` elements to register the following aggregation functions on the non-key fields of `price` and `quant`:

```
<output>
    <field-expr>ESP_aLast(price)</field-expr>
    <field-expr>ESP_aLast(quant)</field-expr>
    <field-expr>ESP_aAve(price)</field-expr>
    <field-expr>ESP_aMax(quant)</field-expr>
  </output>
```

Suppose that the following event streams into the Source window:

```
i    n    21    SAIA    0    55110    932    16.2328    100    4    92    3
58    1
```

The following event is subsequently written by the Aggregate window:

```
I    N     SAIA    16.2328    100    16.2328    100
```

Later, the following event streams into the Source window:

```
i    n    392    SAIA    7    55110    934    15.1296    400    3    64    5
75    1
```

The following event is then written by the Aggregate window:

```
U    N    SAIA    15.1296    400    15.6812    400
```

Lastly, the following event streams into the Source window:

```
i    n    531    SAIA    1    55110    934    15.2872    1100    5    52    73
82    0
```

The following event is then by the Aggregate window:

```
U    N    SAIA    15.2872    1100    15.549867    1100
```

By using `ESP_aLast(fieldName)` and then adding the aggregate fields of interest, you can avoid the subsequent Join window. This makes the modeling cleaner.

Using Compute Windows

Overview to Compute Windows

Use a Compute window to enable a one-to-one transformation of input events to output events through the computational manipulation of the input event stream fields. You can use the Compute window to project input fields from one event to a new event and to augment the new event with fields that result from a calculation.

The set of key fields can be changed within the Compute window, but use this capability with caution. When you make a key field change within the compute window, the Inserts, Updates, and Deletes for the input events' keys must be equivalent Inserts, Update, and Deletes for the new key set.

For information about the XML elements associated with Compute windows, see "window-compute" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* and "XML Language Elements Relevant to Compute and Filter Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Using Compute Functions

Events that enter a Compute window are placed into a group based on the Compute window's key fields. Functions or expressions are run on each group to compute each non-key field of the Compute window.

Compute windows perform computations on input streams using expressions, user-defined functions, or plug-in functions. Output fields from Compute windows can be pushed to another window or to a subscribe connector.

User-defined functions are specified in the **udf** of the **udfs** and **expr-initialize** elements at the beginning of **window-compute**. For an example, see "Using Splitter Functions in Expression Engine Language".

Registered plug-in functions are specified in the **field-plug** XML language element within the **output** element of **window-compute**. These functions are sourced from a shared library, such as **libmethod.so** or **libmethod.dll** found in the **DFESP_HOME** directory.

Using Union Windows

A Union window unites two or more event streams using a strict policy or a loose policy. For information about the XML elements associated with a Union window, see "window-union" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

All input windows to a Union window must have the same schema. The default value of the strict flag is **true**, which means that the key merge from each window must semantically merge cleanly.

The following table explains the results within the Union window given various combinations of state and the value of the strict flag.

Table 3.7 *Results within the Union Window*

State of Union Window	Value of strict	Union Window Result
stateless	**false**	Inserts are replaced with Upserts. Deletes are replaced with safe Deletes
stateless	**true**	Insert and Delete records pass unchanged.
stateful	**false**	Inserts with duplicated keys are processed as Upserts. Deletes with non-existing keys are processed as safe Deletes.
stateful	**true**	Inserts with duplicated keys and Deletes with non-existing keys generate an error and are processed as bad records.

Using Join Windows

Overview to Join Windows

A Join window receives events from a left input window and a right input window. It produces a single output stream of joined events. Joined events are created according to a user-specified join type and user-defined join conditions.

The join order is determined at the edge level of the project. The left window is the first window that is defined as a connecting edge to the Join window. The second window that is defined as a connecting edge is the right window.

Using XML, you can explicitly assign a role to the edge to define which window connects to the Join window: **`left`** or **`right`**. For example:

```
<edges>
   <edge source='w_source1' target='w_join' role='left'/>
   <edge source='w_source2' target='w_join' role='right'/>
</edges>
```

There are some restrictions placed on the types of joins that can be used. The four join types include:

left-outer join
: A left-outer join produces joined output events for every event that arrives from the left window. Joined events are created even when there are no matching events from the right window.

right-outer join
: A right-outer join produces joined output events for every event that arrives from the right window. Joined events are created even when there are no matching events from the left window.

inner join
: An inner join creates joined events only when there is one or more matching events on the side opposite of the input event.

full outer join
: A full outer join creates joined events for every event that arrives from the left window or right window of the joins. Output is always produced.

User-defined join conditions specify what key fields from the left and right input windows are used to generate joined events. Join conditions are an n-tuple of equality expressions. Each expression involves one field from the left window and one field from the right. For example: *(left.f_1 = = right.f_{10}), (left.f_2 = = right.f_7), ... (left.$field_{10}$ == right.$field_1$)* .

To calculate the join non-key fields when new input events arrive, a Join window takes one of the following:

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

For information about the XML elements associated with a Join window, see "window-join" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* and "XML Language Elements Relevant to Join Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Understanding Streaming Joins

Overview to Streaming Joins

The following definition is essential to understanding streaming joins: an X-to-Y join is a join where the following holds:

- a single event from the left window can effect at most X events in the Join window
- a single event in the right window can effect at most Y events in the Join window.

Given a left window, a right window, and a set of join conditions, a streaming join can be classified into one of three different categories.

Join Category	Description
one-to-one joins	An event on either side of the join can match at most one event from the other side of the join. This type of join always involves two dimension tables.
one-to-many joins (or many-to-one joins)	An event that arrives on one side of the join can match many rows of the join. An event that arrives on the other side of the join can match at most one row of the join. In order for a join to be classified as a one-to-many (or many-to-one) join, the following must be true: - a change to one side of the join can affect many rows - a change to the other side can affect at most one row
many-to-many joins	A single event that arrives on either side of the join can match more than one event on the other side of the join.

In a streaming context, every window has a primary key that enables the insertion, deletion, and updating of events. The key fields of a Join window are derived from its input windows and are never specified by the user. When an event arrives on

either side, you must be able to compute how the join changes, given the nature of the arriving data (Insert, Update, or Delete).

SAS Event Stream Processing determines the keys for a Join window based on the join type and join conditions that are specified by the user. SAS Event Stream Processing follows a set of axioms to maintain consistency for the most common join cases:

Join Category	Key Derivation Axiom
one-to-one joins	For a left-outer join, the keys of the left input window are passed through to the Join window. The keys of the right window are never passed to the Join window. This is because a Join window event is produced when a left window event arrives, even if there is no matching right window event. For a right-outer join, the keys of the right input window are passed through to the Join window. The keys of the left window are never passed to the Join window. This is because a Join window event is produced when a right window event arrives, even if there is no matching left window event.
one-to-many joins (or many-to-one joins)	For a one-to-many and many-to-one join, the keys of the "many side" are used as the Join window key. In order to distinguish between Join window events, the many side is the side without all its keys specified in the join conditions.
many-to-many joins	For a many-to-many join, the union of the keys of the left and right input windows are used as the key fields for the Join window. Because a left window event might match multiple right window events, the keys from the right window are needed to distinguish between Join window events. Likewise, because a right window event might match multiple left window events, the keys from the left window are needed to distinguish between Join window events.

Left and right input windows are classified as dimension windows or fact windows. Dimension windows are those whose entire set of key fields participate in the join conditions. Fact windows are those that have at least one key field that does not participate in the join conditions. Input windows are not classified as dimension or fact windows before sending events to a join window.

The following table summarizes the allowed join sub-types and key derivation based on the axioms and the specified join conditions.

Join Category	Left Window	Right Window	Allowed Type	Key Derivation	Streaming Window
one-to-one	dimension	dimension	left-outer	Join keys are keys of left window.	left

Join Category	Left Window	Right Window	Allowed Type	Key Derivation	Streaming Window
			right-outer	Join keys are keys of right window.	right
			full outer	Join keys are keys of left window (arbitrary choice).	left
			inner	Join keys are keys of left window (arbitrary choice).	left
one-to-many	fact	dimension	left-outer	Join keys are keys of left window (right window is lookup).	left
			inner	Join keys are keys of left window (right window is lookup).	left
many-to-one	dimension	fact	right-outer	Join keys are keys of right window (left window is lookup).	right
			inner	Join keys are keys of right window (left window is lookup).	right
many-to-many	fact	fact	inner	Join keys are the full set of keys from the left and right windows.	none

Note: When all the keys of a window are used in a join condition, adding additional non-key fields on the side of the condition is not honored. For example, suppose that the set of left-hand fields that participate in a join condition contain all of the keys of the left window. Any non-key fields in that set are ignored.

Using Secondary Indexes

For allowed one-to-many and many-to-one joins, a change to the fact table enables immediate lookup of the matching record in the dimension table through its primary index. All key values of the dimension table are mapped in the join conditions. However, a change to the dimension table does not include a single primary key for a matching record in the fact table. This illustrates the many-to-one nature of the join. By default, matching records in the fact table are sought through a table scan.

For very limited changes to the dimension table there is no additional secondary index maintenance, so the join processing can be optimized. Here, the dimension table is a static lookup table that can be pre-loaded. All subsequent changes happen on the fact table.

When a large number of changes are possible to the dimension table, it is suggested to enable a secondary index on the join. Automatic secondary index generation is enabled by specifying a join parameter when you construct a new Join window. This causes a secondary index to be generated and maintained automatically when the join type involves a dimension table.

There is a slight performance penalty when you run with secondary indexes turned on. The index needs to be maintained with every update to the fact table. But generating a secondary index has the advantage of eliminating all table scans when changes are made to the dimension table. Secondary index maintenance is insignificant compared with elimination of table scans. With large tables, you can achieve time savings of two to three orders of magnitude by using secondary indexes.

For many-to-many joins, it is recommended to enable secondary indexes.

Using Regeneration versus No Regeneration

The default join behavior is to always regenerate the appropriate rows of a Join window when a change is made to either side of the joins. The classic example of this is a left outer join: the right window is the lookup window, and the left table is the fact (streaming) window. The lookup side of the join is usually pre-populated, and as events stream though the left window, they are matched and the joined events output. Typically, this is a one-to-one relation for the streaming side of the join: one event in, one combined event out. Sometimes a change is made on the dimension side. This change can be in the form of an update to an event, a deletion of an event, or an insertion of a new event. The default behavior is to issue a change set of events that keeps the join consistent.

In regeneration mode, the behavior of a left outer join on a change to the right window (lookup side) is as follows:

- Insert: find all existing fact events that match the new event. If any are found, issue an update for each of these events. They would have used nulls for fields of the lookup side when they were previously processed
- Delete: find fact events that match the event to be deleted. If any are found, issue an update for each of these events. They would have used matching field values for the lookup event, and now they need to use nulls as the lookup event is removed.
- Update: Behaves like a delete of the old event followed by an insert of the new event. Any of the non-key fields of the lookup side that map to keys of the streaming side are taken into account. It is determined whether any of these fields changed value.

With no-regeneration mode, when there is a left outer join on a change to the right window (lookup side), changes to the dimension (lookup) table affect only new fact events. All previous fact events that have been processed by the join are not regenerated. This frequently occurs when a new dimension window is periodically flushed and re-loaded.

The Join window has a `no-regenerates` flag that is false by default. This gives the join full-relational join semantics. Setting this flag to true for your Join window enables the `no-regenerates` semantics. Setting the flag to true is permitted for any of the left or right outer joins, along with one-to-many, many-to-one, and one-to-one inner joins. When a Join window is running in `no-regenerates` mode, it optimizes memory usage by omitting the reference-counted copy of the fact window's index that is normally maintained in the Join window.

Creating Empty Index Joins

Suppose there is a lookup table and an insert-only fact stream. You want to match the fact stream against the lookup table (generating an Insert) and pass the stream out of the join for further processing. In this case, the join does not need to store any fact data. Because no fact data is stored, any changes to the dimension data affect only subsequent rows. The changes cannot go back through existing fact data (because the join is stateless) and issue updates. You must enable the `no-regenerates` property to ensure that the join does not try to go back through existing data.

Suppose there is a join of type LEFT_OUTER or RIGHT_OUTER. The index type is set to pi_EMPTY, rendering a stateless Join window. The `no-regenerates` flag is set to TRUE. This is as lightweight a join as possible. The only retained data in the join is a local reference-counted copy of the dimensions table data. This copy is used to perform lookups as the fact data flows into, and then out of, the join.

On a Join window, you cannot specify insert-only for left and right inputs independently. Specifying insert-only for both sides of the join by setting the Join window to "insert only" is too restrictive. This would not permit changes to the lookup, or non-streaming side of the join. You must follow these rules to ensure expected results.

- A many-to-many join cannot have an empty index.
- The streaming side of a join, as specified in the join classification table, can receive only inserts.

Examples of Join Windows

The following example shows a left outer join. The left window processes fact events and the right window processes dimension events.

```
left input schema:
"ID*:int32,symbol:string,price:double,quantity:int32,
                          traderID:int32"

right input schema: "tID*:int32,name:string"
```

Your code for the Join window would look like this:

```
<window-join name='myJoinWindow' index='pi_RBTREE'>
   <join type='leftouter'>
        <conditions>
         <fields left='traderID' right='tID'/>
        </conditions>
   </join>
  <output>
        <field-selection name='sym' source='l_symbol'/>
        <field-selection name='price' source='l_price'/>
        <field-selection name='tID' source='l_traderID'/>
        <field-selection name='traderName' source='r_name'/>
  </output>
</window-join>
```

Note the following:

- Join conditions take the following form. They specify what fields from the left and right events are used to generate matches.

  "l_*fieldname*=r_*fieldname*, ...,l_*fieldname*=r_*fieldname*"

- Join selection takes the following form. It specifies the list of non-key fields that are included in the events generated by the Join window.

  "{l|r}_*fieldname*, ...{l|r}_*fieldname*"

- Field signatures take the following form. They specify the names and types of the non-key fields of the output events. The types can be inferred from the fields specified in the join selection. However, when using expressions or user-written functions (in C++), the type specification cannot be inferred, so it is required:

  "*fieldname*:*fieldtype*, ..., *fieldname*:*fieldtype*

When you use non-key field calculation expressions, your code looks like this:

```
<window-join name='myJoinWindow' index='pi_RBTREE'>
  <join type='leftouter'>
     <conditions>
```

```
            <fields left='traderID' right='tID'/>
        </conditions>
    </join>
    <output>
          <field-expr name='sym' type='string'>l_symbol</field-expr>
          <field-expr name='price' type='double'>l_price</field-expr>
          <field-expr name='tID' type='int32'>l_traderID</field-expr>
          <field-expr name='traderName' type='string'>r_name</field-expr>
    </output>
</window-join>
```

This shows one-to-one mapping of input fields to join non-key fields. You can use calculation expressions and functions to generate the non-key join fields using arbitrarily complex combinations of the input fields.

Using Transpose Windows

You can conceptualize an event as a row that consists of multiple columns. Use a Transpose window to interchange an event's rows as columns, or its columns as rows. Use attributes of the Transpose window to govern the rearrangement of data.

For information about the XML elements associated with a Transpose window, see "window-transpose" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

The Transpose window accepts only Insert events, and it always has an index of **pi_EMPTY**. It provides two modes: wide and long.

Table 3.8 *Transpose Window Modes*

Mode	Description
wide	Produces one event per incoming event.
long	Produces one or more event per incoming event.

Suppose that you are using wide mode to process information about the pitch, yaw, roll, and velocity of an aircraft in flight. The Source window uses the following schema:

```
<schema>
      <fields>
          <field name='ID'        type='int64' key='true'/>
          <field name='PlaneID'   type='string'/> <!--1-->
          <field name='TAG'       type='string'/> <!--2 -->
          <field name='value'     type='double'/> <!--3 -->
          <field name='time'      type='stamp'/>
          <field name='lat'       type='double'/> <!--4 -->
          <field name='long'      type='double'/>
      </fields>
```

```
</schema>
```

1 The value of the `PlaneID` field of the schema identifies which aircraft provides the data.

2 The value of the `TAG` field specifies whether the data contains the aircraft's pitch, yaw, roll, or velocity. The `TAG` field must be type string.

3 The `value` field records the value for the `TAG` and the specific `time` that the data is recorded.

4 The event captures the plane's latitude (`lat`) and longitude (`long`) when the pitch, yaw, roll, or velocity is recorded.

Now suppose that the following events stream through the Source window.

```
i,n,1,turboprop #1, pitch,1.1, 2017-08-30 15:21:00.000000,10,10
i,n,2,turboprop #2, velocity,-1.2, 2017-08-30 15:21:00.000001,20,20
i,n,3,turboprop #1, roll,-1.1, 2017-08-30 15:21:00.000002,11,11
i,n,4,turboprop #2, yaw,2.2, 2017-08-30 15:21:00.000003,21,21
i,n,5,turboprop #2, pitch,1.2, 2017-08-30 15:21:00.000004,22,22
i,n,6,turboprop #1, velocity,-1.1, 2017-08-30 15:21:00.000005,12,12
i,n,7,turboprop #2, roll,-1.2, 2017-08-30 15:21:00.000006,23,23
i,n,8,turboprop #1, yaw,2.1, 2017-08-30 15:21:00.000007,13,13
i,n,9,jet #1, pitch,1.3, 2017-08-30 15:21:00.000012,30,30
i,n,10,jet #2, velocity,-4.4, 2017-08-30 15:21:00.000013,40,40
i,n,11,jet #1, roll,-4.3, 2017-08-30 15:21:00.000014,31,31
i,n,12,jet #2, yaw,8.4, 2017-08-30 15:21:00.000015,41,41
i,n,13,jet #2, pitch,4.4, 2017-08-30 15:21:00.000016,42,42
i,n,14,jet #1, velocity,-4.3, 2017-08-30 15:21:00.000017,32,32
i,n,15,jet #2, roll,-4.4, 2017-08-30 15:21:00.000018,43,43
i,n,16,jet #1, yaw,8.3, 2017-08-30 15:21:00.000019,33,33
i,n,17,turboprop #1, pitch,23.1, 2017-08-30 15:21:06.000022,14,14
```

In these seventeen events, four planes stream data through the Source window: **`turboprop #1`**, **`turboprop #2`**, **`jet #1`**, and **`jet #2`**. Each event that streams through contains either a pitch, velocity, roll, or yaw value at a specific time.

Now suppose you stream the input data through a Transpose window. Using `mode='wide'`, you create a single event (row) that contains the pitch, velocity, roll, and yaw of each aircraft at a specific time.

```
<window-transpose name='transposeW'
                        mode='wide' <!-- 1 -->
                        tag-name='TAG'<!-- 2 -->
                        tags-included='pitch,yaw,roll,velocity'
<!-- 3 -->
                        tag-values='value,time' <!-- 4 -->
                        clear-timeout='never'
                        group-by='PlaneID'> <!-- 5 -->
     <connectors>
      ...
     </connectors>
</window-transpose>
```

1 Using `wide` mode produces output events that contain multiple columns, each based on data streamed through the input events.

2 The values of `TAG` in the input events determine the columns in output events.

3 `Tags-included` specify which specific values of `TAG` are to be included in the output events. Here, all four values of `TAG` are specified.

4 The `value` and `time` that are associated with pitch, yaw, roll, and velocity are included in the output event. The associated latitude and longitude are passed through, and not included.

5 Output events are grouped by the value of `PlaneID`.

Output columns are formed by taking the cross product of the following:

{`pitch`, `yaw`, `roll`, `velocity`} times {`value`, `time`}

This yields `pitch_value`, `pitch_time`, `yaw_value`, `yaw_time`, and so on.

Here are the first four events that stream from the Transpose window after processing events four events from the Source window:

```
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">1</value>
  <value name="PlaneID">turboprop #1</value>
  <value name="lat">10.000000</value>
  <value name="long">10.000000</value>
  <value name="pitch_time">1504106460000000</value>
  <value name="pitch_value">1.100000</value>
</event> <!-- 1 -->
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">2</value>
  <value name="PlaneID">turboprop #2</value>
  <value name="lat">20.000000</value>
  <value name="long">20.000000</value>
  <value name="velocity_time">1504106460000001</value>
  <value name="velocity_value">-1.200000</value>
</event> <!-- 2 -->
$ <event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">3</value>
  <value name="PlaneID">turboprop #1</value>
  <value name="lat">11.000000</value>
  <value name="long">11.000000</value>
  <value name="pitch_time">1504106460000000</value>
  <value name="pitch_value">1.100000</value>
  <value name="roll_time">1504106460000002</value>
  <value name="roll_value">-1.100000</value>
</event> <!-- 3 -->
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">4</value>
  <value name="PlaneID">turboprop #2</value>
  <value name="lat">21.000000</value>
  <value name="long">21.000000</value>
  <value name="velocity_time">1504106460000001</value>
  <value name="velocity_value">-1.200000</value>
  <value name="yaw_time">1504106460000003</value>
  <value name="yaw_value">2.200000</value>
</event> <!-- 4 -->
```

1 The first input event contains a pitch value of 1.1 for **`turboprop #1`**.

In this output event and all subsequent output events, the Transpose window passes through `lat` and `long` unchanged.

Because TAG has the value **pitch** and time and value are the tags-included, the new variables pitch_time and pitch_value are generated. The values of pitch_time and pitch_value are inserted.

There are no values of roll, velocity, or yaw to process in the first event.

2 The second input event contains a velocity value of –1.2 for **turboprop #2**. Because TAG has the value **velocity** and time and value are the tags-included, the new variables velocity_time and velocity_value are generated.

There are no values of pitch, roll, or yaw to process in the second event.

3 The third input event contains a roll value of –1.1 for **turboprop #1**.

The plane's pitch_time and pitch_value are retained from the first event.

Because TAG has the value **roll** and time and value are the tags-included, the new variables roll_time and roll_value are generated.

There are no values of velocity or yaw to process in the third event.

4 The fourth input event contains a yaw value of 2.2 for **turboprop #2**.

The plane's velocity_time and velocity_value are retained from the second event.

Because TAG has the value **yaw** and time and value are the tags-included, the new variables yaw_time and yaw_value are generated.

There are no values of pitch or roll to process in the fourth event.

In this way, the Transpose window builds an event that contains every TAG value of interest per plane, because **PlaneID** is the value of group-by. By the time that all seventeen input events are processed, there are four events that capture each plane's pitch, roll, velocity, and yaw:

```
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">7</value>
  <value name="PlaneID">turboprop #2</value>
  <value name="lat">23.000000</value>
  <value name="long">23.000000</value>
  <value name="pitch_time">1504106460000004</value>
  <value name="pitch_value">1.200000</value>
  <value name="roll_time">1504106460000006</value>
  <value name="roll_value">-1.200000</value>
  <value name="velocity_time">1504106460000001</value>
  <value name="velocity_value">-1.200000</value>
  <value name="yaw_time">1504106460000003</value>
  <value name="yaw_value">2.200000</value>
</event>
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">8</value>
  <value name="PlaneID">turboprop #1</value>
  <value name="lat">13.000000</value>
  <value name="long">13.000000</value>
  <value name="pitch_time">1504106460000000</value>
  <value name="pitch_value">1.100000</value>
  <value name="roll_time">1504106460000002</value>
  <value name="roll_value">-1.100000</value>
  <value name="velocity_time">1504106460000005</value>
  <value name="velocity_value">-1.100000</value>
```

```
  <value name="yaw_time">1504106460000007</value>
  <value name="yaw_value">2.100000</value>
</event>
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">15</value>
  <value name="PlaneID">jet #2</value>
  <value name="lat">43.000000</value>
  <value name="long">43.000000</value>
  <value name="pitch_time">1504106460000016</value>
  <value name="pitch_value">4.400000</value>
  <value name="roll_time">1504106460000018</value>
  <value name="roll_value">-4.400000</value>
  <value name="velocity_time">1504106460000013</value>
  <value name="velocity_value">-4.400000</value>
  <value name="yaw_time">1504106460000015</value>
  <value name="yaw_value">8.400000</value>
</event>
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">16</value>
  <value name="PlaneID">jet #1</value>
  <value name="lat">33.000000</value>
  <value name="long">33.000000</value>
  <value name="pitch_time">1504106460000012</value>
  <value name="pitch_value">1.300000</value>
  <value name="roll_time">1504106460000014</value>
  <value name="roll_value">-4.300000</value>
  <value name="velocity_time">1504106460000017</value>
  <value name="velocity_value">-4.300000</value>
  <value name="yaw_time">1504106460000019</value>
  <value name="yaw_value">8.300000</value>
</event>
```

The last input event updates the pitch of **`turboprop #1`**, which was collected at a later value of `time`.

```
<event opcode="insert" window="newproject/cq/transposeW">
  <value name="ID">17</value>
  <value name="PlaneID">turboprop #1</value>
  <value name="lat">14.000000</value>
  <value name="long">14.000000</value>
  <value name="pitch_time">1504106466000022</value>
  <value name="pitch_value">23.100000</value>
  <value name="roll_time">1504106460000002</value>
  <value name="roll_value">-1.100000</value>
  <value name="velocity_time">1504106460000005</value>
  <value name="velocity_value">-1.100000</value>
  <value name="yaw_time">1504106460000007</value>
  <value name="yaw_value">2.100000</value>
</event>
```

Subsequent input events continue to update each plane's output event.

Use the `clear-timeout` attribute of the Transpose window to specify a time after which output event values clear unless an input event value is received.

Suppose that input data does not arrive from multiple devices or pieces of equipment (such as from two planes in the previous example). In that case, you do not need to use the `group-by` attribute.

For example, consider the following input data. The Source window uses the same schema as before, but without `PlaneID`.

```
i,n,1,velocity, 234.0
i,n,2,roll,2.3
i,n,3,pitch,3.4
i,n,4,yaw,-1.1
```

You could transpose this data by specifying a Transpose window with the following attributes:

```
<window-transpose name='transposeW'
         mode='wide'
         tag-name='TAG'
         tags-included='pitch,yaw,roll,velocity'
         tag-values='value'
         clear-timeout='never'
/>
```

IMPORTANT The Transpose window in wide mode uses the `group-by` field to group events. It requires the cardinality of the `group-by` field to be bounded. If the field is not bounded, then the window continues to consume memory with no bounds.

When you use long mode, you obtain the inverse results of wide mode. The Transpose window streams a number of events for each wide event that it receives. Input schema for the Source window must reflect combinations of fields.

For example, consider the schema of this Source window:

```
<schema>
                <fields>
                  <field name='ID'          type='int64' key='true'/>
                  <field name='PlaneID'     type='string'/>
                  <field name='pitch_value' type='double'/>
                  <field name='pitch_time'  type='stamp'/>
                  <field name='yaw_value'   type='double'/>
                  <field name='yaw_time'    type='stamp'/>
                  <field name='roll_value'  type='double'/>
                  <field name='roll_time'   type='stamp'/>
                  <field name='velocity_value' type='double'/>
                  <field name='velocity_time'  type='stamp'/>
                  <field name='lat'       type='double'/>
                  <field name='long'      type='double'/>
                </fields>
              </schema>
```

Suppose that you stream a wide event through that Source window.

```
I N 1 turboprop #2 1.2 21:00.0 2.2 21:00.0 -1.2 21:00.0 -1.2 21:00.0
20 20
```

A downstream Transpose window looks up combinations of `tag-values` and `tags-included` values in the incoming schema.

```
<window-transpose name='transposeL'
                            mode='long' tag-name='TAG'
                            tag-values='value,time'
```

```
                                tags-included='pitch,yaw,roll,velocity'>
                ...           </window-transpose>
```

Processing that input event, the Transpose window streams the following four output events:

```
I,N, 1,0,pitch,1.200000,2017-08-30 15:21:00.000004,turboprop
#2,20.000000,20.000000
I,N, 1,1,yaw,2.200000,2017-08-30 15:21:00.000003,turboprop
#2,20.000000,20.000000
I,N, 1,2,roll,-1.200000,2017-08-30 15:21:00.000006,turboprop
#2,20.000000,20.000000
I,N, 1,3,velocity,-1.200000,2017-08-30 15:21:00.000001,turboprop
#2,20.000000,20.000000
```

If the downstream Transpose window does not find the appropriate combinations of `tag-values` and `tags-included` values, the window fails in finalization and does not permit the model to run.

Using Copy Windows

Overview to Copy Windows

Use a Copy windows to copy a parent window of any other type. They are useful to retain events with specified event state retention policies. You can set retention policies only in Source and Copy windows.

For information about the XML elements associated with Copy windows, see "window-copy" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Retention Policies in Copy Windows

You can set event state retention for a Copy window only when the window is not specified to be insert-only and when the window index is not set to `pi_EMPTY`. All subsequent sibling windows are affected by retention management. Events are deleted when they exceed the windows retention policy.

For more information about retention policies, see "Understanding Retention".

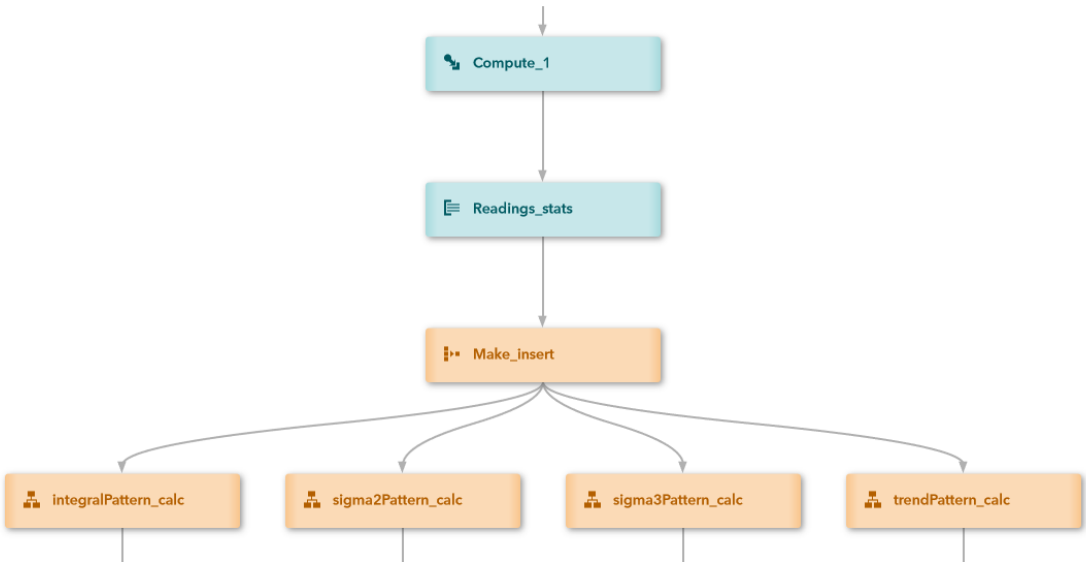
Using Remove State Windows

Use a Remove State window to facilitate the transition of a stateful part of a model to a stateless part of a model. A Remove State window converts all events that it receives into Inserts and adds a field named `eventNumber`, which is a monotone-increasing sequential integer. This added field is the only key of the Remove State window.

For information about the XML elements associated with Remove State windows, see "window-remove-state" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Consider the following model:

Figure 3.2 *Stateful Windows Streaming into Stateless Pattern Windows*



Compute_1, a Compute window, receives a stream of sensor and location data. It augments incoming events with new fields that are based on a manipulation of the incoming data.

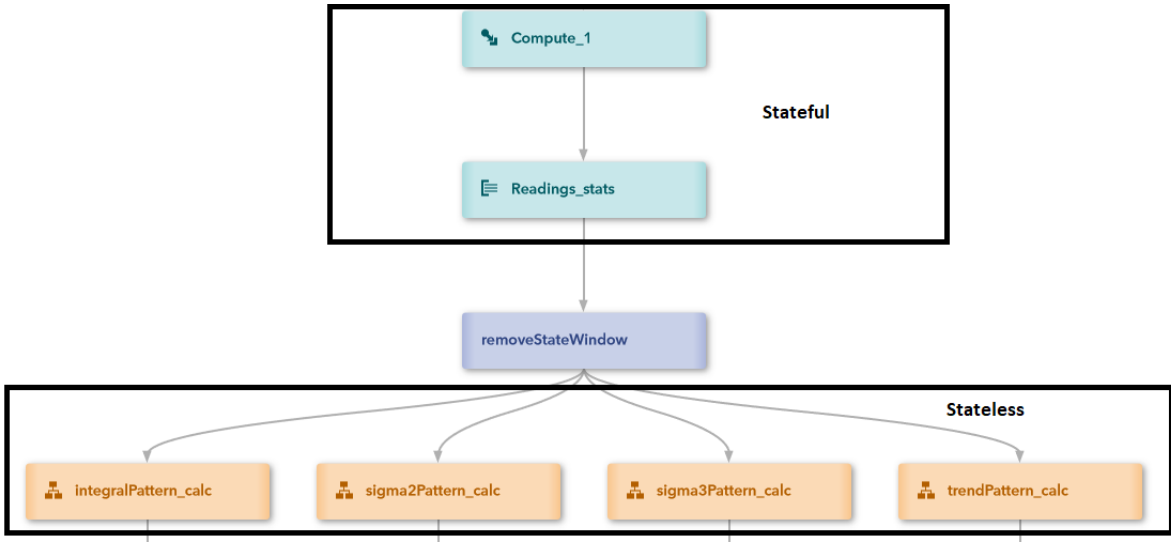
Compute_1 streams augmented events into Readings_stats, an Aggregate window. Readings_stats groups events by a location and calculates values such as sum, average, standard deviation, and so on, for the group. Every time that a new event for a specific location streams into Reading_stats, an Update event that contains these calculated values streams to Make_insert , a Calculate window.

Make_insert contains a DS2 function that converts Update events into Insert events.

Make_insert streams those Insert events to a series of Pattern windows, which accept only Insert events. These Pattern windows further process the data.

In this model, the only purpose of the Calculate window is to convert Update events into Insert events. You can replace Make_insert with a Remove State window to accomplish the same result.

Figure 3.3 *Using the Remove State Window*



The generic form of the Remove State window schema is as follows:

```
eventNumber*:int64, [originalOC:string, originalFL:string,] <incoming_schema>
```

The following XML code shows a common case. The *incoming_schema* is as follows:

```
symbol:string, true_test:integer
```

```
<window-remove-state name='removeStateWindow' add-log-fields='true'
<!--1-->
       remove='retentionUpdates retentionDeletes'><!--2-->
</window-remove-state>
```

1 Setting `add-log-fields='true'` adds the originalOC and originalFL fields specified in the schema.

2 You can configure the Remove State window to filter events by opcode or retention policy or combination of the two. Setting `remove='retentionDeletes retentionUpdates'` filters out Delete and Update events that have the retention flag set.

Sample output from this Remove State window is as follows. Three events are filtered:

```
<event opcode="insert" window="project/contQuery/removeStateWindow">
  <value name="eventNumber">13</value> <!--1 -->
  <value name="originalFL">N</value> <!--2 -->
  <value name="originalOC">D</value> <!-- 3-->
  <value name="symbol">id</value>
  <value name="true_test">21</value>
</event>
<event opcode="insert" window="project/contQuery/removeStateWindow">
  <value name="eventNumber">14</value>
  <value name="originalFL">N</value>
  <value name="originalOC">D</value>
```

```
  <value name="symbol">pd</value>
  <value name="true_test">23</value>
</event>
<event opcode="insert" window="project/contQuery/removeStateWindow">
  <value name="eventNumber">15</value>
  <value name="originalFL">N</value>
  <value name="originalOC">D</value>
  <value name="symbol">pd</value>
  <value name="true_test">23</value>
</event>
```

1 The `eventNumber` is inserted by the Remove State window into each event that passes through.

2 The original flag of the event number 13 was N (normal).

3 The original opcode of the event number 13 was D (Delete).

Using Functional Windows

Overview to Functional Windows

Use a Functional window to define entities in order to manipulate or transform event data. Within a Functional window, you define a schema and then a *function context* to apply to the event data. You set the function context one of three ways:

- Specify a regular expression.
- Specify properties of connectors, functions, projects, Procedural windows, or streaming analytics windows.
- Use a function, which can include a supporting function.

For information about the XML elements associated with Functional windows, see:

- "window-functional" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*
- "XML Language Elements Relevant to Functional Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*

Generating Events

Use the XML element `<generate>` to specify a Boolean function or expression that determines whether to include original events with output events:

- If the function or expression evaluates to true, then the original event is included with any additional output events.
- If it evaluates to false, then the original event is not included.

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

Alternatively, you can generate events within a Lua window. For more information, see Generate Events from JSON Using the Lua Window.

Using Event Loops

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

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

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

The event loop creates a contextual XML object for each iteration. You can refer to this object as **`#_context`** within a function. The namespace of this object is set to that of the top-level container. If you use a string representation instead of the **`#_context`** object, you cannot set the namespace.

Understanding and Using Function Context

Specifying Expressions

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

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

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

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

Note: POSIX regular expressions must follow the standards specified by the IEEE.

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

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

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

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

Specifying Properties

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

There are five types of properties:

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

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

Table 3.9 How to Code Each Property Type

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

Suppose you had some employee information streaming into the model.

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

You can create a `property-map` to store employee data and then examine the position field to create a `property-xml` containing the appropriate data. If the employee is a developer, the XML is from `developerInfo`. Otherwise, it uses `managerInfo`. Your function-context would look like this, where the functions are coded using the if,equals,mapValue, and xpath support functions:

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

Streaming in the following event:

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

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

Yields the following result:

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

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

Using Functions

When you use a function, the Functional window processes events looking for a function with a name that corresponds to each field in its schema. If the function exists, then the window runs it. The resulting value is entered into an output event. If no function is specified for a field and a field with the same name exists in the input schema, then the input value is copied directly into the output event.

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

- general functions
- functions specific to event stream processing

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

Table 3.10 *Data Mappings Relevant to Using Functions in a Function Context*

string	ESP_UTF8STR
float	ESP_DOUBLE
long	ESP_INT64
integer	ESP_INT32
Boolean	ESP_INT32

For example, suppose that you have a `name` field in the input event and you want to generate an `occupation` field in the output event. You could code the function as follows using the ifNext andequals support functions:

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

```
</function>
```

You can also reference fields in the output event. Continuing with this example, perhaps you want to add the `hourlyWage` field to the output event depending on the value of `occupation`, again using the ifNext and equals support functions:

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

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

When the Functional window comes across a reference to an event field, it first looks in the output event (the event that is being generated) for a value. If no value is found in this event, the Functional looks for a value in the input event. The values in the output event are generated in the order they appear in the XML code, so any fields defined after another field can reference its value in the function.

Using Lua Windows

Overview of Lua Windows

The Lua window enables you to generate SAS Event Stream Processing events with Lua programs. Within a Lua window, you represent an SAS Event Stream Processing input event by a table. To create multiple output events from a single input event, you put those events into an indexed array.

You can debug your Lua code as you design your window. The Lua `print()` function prints output to the console. If you use this function in your code, then you see the output in the *ESP server* console window.

For information about additional Lua functionality available through the Lua window, see Chapter 8, “Common Lua Functionality”.

IMPORTANT Your Lua code can include any function provided in the standard Lua libraries with the following two exceptions:

- `os.execute`
- `io.popen`

The use of either of these functions results in a runtime error, which prevents a project from being loaded.

Configuring a Lua Window

Here is sample XML code to specify a Lua window:

```
<window-lua name="name" index="index" events="create_function" init=""
destroy="" heartbeat="hb_function" process-blocks="" encode-binary=""
on-script-exception="log-and-continue | stop-and-remove"> <!--1-->
  <schema> <!--2-->
    <fields>
      <field name="name" type="type" key="key"/>
      ...
    </fields>
  </schema>
  <copy exclude="true|false">...</copy> <!--3-->
  <use exclude="true|false">...</use> <!--4-->
  <code><![CDATA[
   function create_function(data,context) -- 5
        local   e = {}
        ...
        return(e)
    end

   function hb_function(context) -- 6
        ...
   end

  ]]></code>
  <forwarding suppress="true|false">7
     <destinations>
        <destination window="window" contquery="contquery"
blocksize="blocksize"/>
        ...
     </destinations>
  </forwarding>    </window-lua>
```

1 The top level element, `<window-lua>`, identifies this as a Lua window.

The `events` attribute specifies the name of an event creation Lua function that you specify within the `code` element. This function is invoked when the Lua window receives an input event.

The `init` attribute specifies the name of a Lua function to be invoked when the Lua window is initalized.

The `destroy` attribute specifies the name of a Lua function to be invoked when the Lua window is destroyed. You can use this function to perform any Lua cleanup tasks that need to be done.

The `heartbeat` attribute specifies the name of a heartbeat Lua function that you specify within the `code` element. This function is invoked when the Lua

window receives project-level heartbeat messages. This attribute is not required.

The `process-blocks` attribute indicates whether you want to process events in blocks or individually. When you process event blocks (`process-blocks=true`), the event creation function receives an array of Lua tables that represent the events in the block. Otherwise, the function receives a single event (`process-blocks=false`, the default).

The `encode-binary` attribute indicates whether to Base64 encode binary data for event delivery. The default value is **`false`**.

The `on-script-exception` attribute is an optional attribute that specifies how to handle exceptions that are encountered during the execution of window code. When set to log-and-continue, an error is logged and execution continues. When set to stop-and-remove, a fatal error is logged and the project is stopped and removed. If this value is not specified, the window inherits the script handling behavior defined in the containing project.

2 You must specify a schema for the Lua window.

3 When you include the `copy` element, the specified fields are copied from the input event to the output event. Use the `exclude=true` attribute to specify fields not to be copied. When you include an empty `copy` element, all fields are copied to the output event.

4 When you include the `use` element, the specified files from the input event are passed from the *ESP server* into the Lua code. Use the `exclude=true` attribute to specify fields not to be passed. When you include an empty `use` element, no fields are passed into the Lua code.

5 Specify the event creation function to call when the window gets an input event.

6 Specify the heartbeat function to call when the window receives project-level heartbeat messages.

7 When you include the `forwarding` element, events are generated and forwarded to specified destinations. Use the `suppress=true` attribute to prevent event generation in the event forwarding window. For more information, see "Event Forwarding".

The code element can either contain code that is owned by the window, or it can point to shared code that exists in a Lua snippet. If the code is owned by the window it can be either inline or in an external file. If it is in an external file you can use the file attribute to point to the file.

```
<code file="mycode.lua" />
```

```
<code snippet="mysharedcode"/>
```

Note: When referencing a Lua snippet or an external file, the `code` element must not contain code.

When the Lua window uses shared code within a snippet, it shares all data, functions, and variables with any other Lua entity that points to that snippet. The shared code is always locked by the entity that is currently using it.

The parameters for Lua tables are as follows:

- `data` - When the window is processing event blocks, this parameter specifies an array of Lua tables that represent the events in the block. Otherwise, it specifies a single Lua table that represents a single event. In both cases, input events include the fields specified in `use`, or all fields when `use` is not specified.
- `context` - Context data for the current event. Valid keys are as follows:
  - `window` - The window ID of the containing Lua window.
  - `input` - The name of the input window for this event.
  - `counter` - The event count of the Lua window.

The `context` parameter of the heartbeat function is identical to that passed to the event creation function except that you do not specify an input value. You can use a heartbeat function to create events as you would with an events function.

Generating Events

Generating a Single Event

Suppose that you want to generate a single output event per input event. Consider the following Source window:

```
<window-source name="source" insert-only="true">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="string" type="string"/>
      <field name="number" type="int32"/>
      <field name="array" type="array(i32)"/>
    </fields>
  </schema>
</window-source>
```

With an edge between the Source window and the following Lua window, the following Lua window performs these tasks:

- Copies the `id` from the input event to the output event.
- Puts double quotation marks around the `string` field.
- Multiplies the value of `number` by 3.
- Multiplies each value in the `array` by 2.

```
<window-lua name="lua" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="new_string" type="string"/>
      <field name="new_number" type="int32"/>
      <field name="new_array" type="array(i32)"/>
    </fields>
  </schema>
```

```
  <copy>id</copy> <!--1-->
  <code><![CDATA[

    function create(data,context)
        local   e = {} --2
        e.new_string = "\""..data.id.."\"" --3
        e.new_number = data.number * 3 --4
        e.new_array = {} --5
        for index,value in ipairs(data.array) do
            e.new_array[index] = value * 2 --6
        end
        return(e) --7
    end

  ]]></code>
</window-lua>
```

1 The `copy` element specified that the `id` from the input event to the output event.

2 This line of code creates the Lua object to hold the output event data.

3 You need to add the output fields to the Lua object. The `new_string` field is generated with this line of code. Note that the Lua concatenation operator is `'..'`.

4 The `new_number` field is generated by multiplying the input field number by 3.

5 The `new_array` field is an array field. This line of code creates the Lua table.

6 These lines iterate over the values in the input array, multiply each value by 2, and set the new value.

7 This line of code returns the created Lua object.

When the creation function returns nil, no events are created.

For example, the following code does not create events when the input number field is less than 10:

```
function create(data,context)
    if (data.number < 10) then
        return(nil)
    end
    ...
end
```

Given the following input event:

```
i,n,1,string data,33,[11;24;55]
```

The output event from the preceding code looks like this:

```
<event opcode="insert" window="p/cq/lua">
  <value name="id">1</value>
  <value name="new_array">[22;48;110]</value>
  <value name="new_number">99</value>
  <value name="new_string">"string data"</value>
</event>
```

Generating Multiple Events

To generate multiple events, create and return an indexed Lua table. The indexed table contains zero or more Lua tables. Each table represents a set of output events.

For example, consider the following code:

```
function create(data,context)
    local   events = {}

    for i=1,5 do
        local   event = {}
        -- Add event data here
        events[i] = event
    end
    return(events)
end
```

The following Lua window uses this code to generate an event for each entry in the array:

```
<window-lua name="lua" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="new_string" type="string"/>
      <field name="new_number" type="int32"/>
      <field name="array_entry" type="int32"/> <!-- 1 -->
    </fields>
  </schema>
  <code><![CDATA[

    local   eventId = 1

    function create(data,context)
        local   events = {}

        for index,value in ipairs(data.array) do
            local   e = {}

            e.id = esp_toString(eventId)
            eventId = eventId + 1 -- 2

            e.new_string = "\""..data.string.."\""
            e.new_number = data.number * 3
            e.new_array = {}
            e.array_entry = value * 2

            events[index] = e
        end

        return(events)
    end
```

```
  ]]></code>
</window-lua>
```

1 This line results in an output event with single field entries instead of an array.

2 These lines create an indexed array and then iterate over the input array field, generating individual events for each entry. To generate a unique ID for each event, create a local variable named `eventId` and use it to set the output event ID, incrementing it each time after you use it.

The `eventID` variable maintains its value between calls into the Lua code so that you have a unique, incrementing event ID on your output events. Thus, you can use the variable value to maintain state in the Lua window.

Assigning the Event Opcode

By default, the Lua window assigns the opcode of the incoming event to the opcode of the outgoing event or set of events. Alternatively, you can use the `_opcode` field to set a different opcode for the outgoing event or set of events:

```
<code><![CDATA[

    function create(data,context)
        local   e = {}
        e.new_string = "\""..data.id.."\""
        e.new_number = data.number * 3
        e.new_array = {}
        for index,value in ipairs(data.array) do
            e.new_array[index] = value * 2
        end
        e._opcode = "upsert"
        return(e)
    end

  ]]></code>
```

Event Forwarding

To publish events more efficiently, you can use event forwarding, which controls the flow of events through a model. For example, if you have an event block that contains multiple large events, you can set the value of `blocksize` to 1 to break down the block and process events individually.

For example, consider the following code:

```
<forwarding suppress="false"> 1
  <destinations>
      <destination window="sourcewin" contquery="cq_1" blocksize="5"
active="true"/> 2
  </destinations>
</forwarding>
```

1 This line enables event forwarding in a Lua window. The `suppress` attribute ensures that events are forwarded and generated in the window.

2 This line specifies the Source window to which you want to forward events, the name of the continuous query that contains the Source window, and the number of events to put into each event block. This line also enables events to be forwarded to the destination.

Using the RESTful API with Lua Windows

Overview

You can use the RESTful API to communicate with Lua windows in order to perform these tasks:

- call functions that are defined in the Lua code
- add to or change the Lua code in the window
- validate Lua code

Calling Lua Functions

You can call a function within a Lua window by using the following `PUT` request.

```
PUT /windows/project/contquery/window/code?
op=run&function=<function_name>
```

Consider the following code that puts a token into events:

```
local   token = "a_token"

function set_token(data)
    token = data.token
    local   status = {}
    status.code = 0
    status.message = "token has been set to "..token
    return(status)
end
```

You can call the `set_token` function to change the value of the token. Suppose that the window is **`p/cq/lua`**. You can set the token with the following call:

```
PUT /windows/p/cq/lua/code?op=run&function=set_token
```

Suppose you put the following request body into a file named **`data.json`**.

```
{
    "token":"a_token"
}
```

The full request and response would look like this:

```
$ curl "http://{INGRESS_URL}/eventStreamProcessing/v1/windows/p/cq/lua/
code?op=run&function=set_token" --data @data.json -X PUT

{
    "code":"0",
    "message":"token has been set to a_token"
}
```

The *ESP server* transforms the Lua object that is returned by the code into JSON and sends the object back to the client. After this call, you can use the new value of the token in event creation.

Setting Lua Code

You can change the code that runs within a Lua window by using a `PUT` request.

```
PUT /windows/project/contquery/window/code?op=load
```

Suppose that you have a Source window with a numeric ID.

```
<window-source name="s" autogen-key="true" insert-only="true">
  <schema>
    <fields>
      <field name="id" type="int64" key="true"/>
    </fields>
  </schema>
</window-source>
```

Suppose that you want to feed that ID into a Lua window, calculate the sum of that ID and the number 5, and put the result in a numeric field named `out_value`.

```
<window-lua name="lua" events="create">
  <schema-string>id*:int64,out_value:int32</schema-string>
  <code><![CDATA[

    function create(data,context)
        local   event = {}
        event.id = data.id
        event.out_value = data.id + 5
        return(event)
    end

  ]]></code>
</window-lua>
```

Events would look like this code:

```
<event opcode="insert" window="p/cq/lua">
  <value name="id">12</value>
  <value name="out_value">17</value>
</event>
```

Now suppose that you want to add 10 to the ID instead of 5. Create a file called **`code.lua`** that contains your new Lua code.

```
function create(data,context)
    local   event = {}
    event.id = data.id
    event.out_value = data.id + 10
    return(event)
end
```

Load the code through the RESTful API.

```
$ curl "http://{INGRESS_URL}/eventStreamProcessing/v1/windows/p/cq/lua/
code?op=load" -X PUT --data-binary @code.lua

<response>
  <message>Lua code loaded</message>
</response>
```

After this, events would look like this:

```
<event opcode="insert" window="p/cq/lua">
  <value name="id">13</value>
  <value name="out_value">23</value>
</event>
```

Validating Lua Code

You can also use the *ESP server* as a general Lua code validator with the following `POST` request:

```
POST /luaValidationResults
```

When you have Lua code that you want to validate, send it in the request body and receive the results. Suppose that you write the following code and put it in a file named **`code.lua`**.

```
this is not valid Lua code

function create(data,context)
    local   event = {}
    event.id = data.id
    event.out_value = data.id + 10
    return(event)
end
```

Then you run the following command on the console:

```
$ curl "http://{INGRESS_URL}/eventStreamProcessing/v1/
luaValidationResults" -X POST --data-binary @code.lua
```

```
<error errorCode="-1" httpStatusCode="404">
  <message>[string "test"]:2: syntax error near 'is'</message>
  <version>1</version>
</error>
```

Remove the invalid line, and then rerun the command on the console:

```
<response>
  <message>Lua code validated successfully</message>
</response>
```

Using Python Windows

Overview of Python Windows

The Python window enables you to write functions in Python programs that create SAS Event Stream Processing events. The ESP server communicates with Python code through Python objects. Within a Python window, execution of Python programs is single-threaded.

You can use Python windows instead of SAS Micro Analytic Service modules when you work with Python code. Using Python windows provides the following advantages:

- You do not need to create SAS Micro Analytic Service modules and specify input handlers in Calculate windows.
- Debugging code is easier within Python windows than within a SAS Micro Analytic Service module.
- Python windows can deliver better performance when BLOBs are used.
- When you use SAS Event Stream Processing Studio to edit Python code in a Python window, the code editor enables you to validate your Python code.

In Python, a dictionary is a built-in data type that represents an unordered collection of key-value pairs. An event is represented as a Python dictionary whose keys are the schema fields.

You can debug your Python code as you design your window. Use a Python `print()` function to send output to the ESP server console window. For more information, see “Console Debugging”.

Note: When you use the Python window, your code must be compatible with Python 3.11.

Configuring a Python Window

Here is sample XML code to specify a Python window:

```
<window-python name="py_window" index="pi_EMPTY" events="createEvent"
```

```
init="initalize" destroy="cleanup" heartbeat="hb"
process-blocks="false" encode-binary="false"
on-script-exception="log-and-continue | stop-and-remove"> <!--1-->
  <schema> <!--2-->
    <fields>
      <field name="" type="" key=""/>
      ...
    </fields>
  </schema>
  <copy exclude="true|false">...</copy> <!--3 -->
  <use exclude="true|false" expand="true|false">...</use> <!--4 -->
  <code><![CDATA[ # 5
    def initialize():
        ...
        # initialization work
        ...

    def cleanup():
        ...
        # cleanup work
        ...

    def createEvent(data,context): # 6
        e = {}
        ...
        return e
    end

    def hb(context): # 7
        print("got a heartbeat")
  ]]></code>
  <forwarding suppress="true|false">8
     <destinations>
        <destination window="window" contquery="contquery"
blocksize="blocksize"/>
        ...
     </destinations>
  </forwarding></window-python>
```

1 The top-level element, `<window-python>`, identifies this as a Python window.

The `events` attribute specifies the name of an event creation Python function that you specify within the `code` element. This function is invoked when the Python window receives an input event.

The `init` attribute specifies the name of a Python function to be invoked when the Python window is initalized.

The `destroy` attribute specifies the name of a Python function to be invoked when the Python window is destroyed. You can use this function to perform any Python cleanup tasks.

The `heartbeat` attribute specifies the name of a heartbeat Python function that you specify within the `code` element. This function is invoked when the Python window receives project-level heartbeat messages. This attribute is not required. For more information about project-level heartbeat messages, see

"project" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

The `process-blocks` attribute indicates whether you want to process events in blocks or individually. When you process event blocks (`process-blocks=true`), the event creation function receives an array of Python objects that represent the events in the block. Otherwise, the function receives a single event (`process-blocks=false`, which is the default).

The `encode-binary` attribute indicates whether to Base64-encode binary data for event delivery. The default value is **`false`**.

The `on-script-exception` attribute is an optional attribute that specifies how to handle exceptions that are encountered during the execution of window code. When set to log-and-continue, an error is logged and execution continues. When set to stop-and-remove, a fatal error is logged and the project is stopped and removed. If this value is not specified, the window inherits the script handling behavior defined in the containing project.

2 You must specify a schema for the Python window.

3 When you include the `copy` element, the specified fields are copied from the input event to the output event. Use the `exclude=true` attribute to specify fields not to be copied. When you include an empty `copy` element, all fields are copied to the output event.

4 When you include the `use` element, the specified fields from the input event are passed from the ESP server into the Python code. For example, if you have an input event with 100 fields but you use only two of those fields in your code, list those fields in the `use` field. Like the `copy` field, you can set `exclude` to **`true`** so that all fields except those listed are passed into the Python code.

5 Always enclose code in XML `CDATA` notation to enable proper parsing of all characters within the Python code.

6 Specify the event creation function to call when the window receives an input event.

7 Specify the heartbeat function to call when the window receives project-level heartbeat messages.

8 When you include the `forwarding` element, events are generated and forwarded to specified destinations. Use the `suppress=true` attribute to prevent event generation in the event forwarding window. For more information, see "Event Forwarding".

The code element can either contain code, which is owned by the window, or it can point to shared code that exists in a Python snippet. If the code is owned by the window it can be either inline or in an external file. If it is in an external file you can use the file attribute to point to the file.

```
<code file="mycode.py" />
```

```
<code snippet="mysharedcode"/>
```

Note: When referencing a Python snippet or external file, the `code` element must not contain code.

When the Python window uses shared code within a snippet, it shares all data, functions, and variables with any other Python entity that points to that snippet. The shared code is always locked by the entity that is currently using it.

An event creation function is the entry point for the ESP server to call when the window gets an input event. For example, if the event creation function is called `createEvent`, the top-level Python window element would point to that function as follows:

```
<window-python name="python" events="createEvent">
```

An event creation function can take one of two forms. The first form is as follows:

```
<code><![CDATA[
def createEvent(data,context):
     e = {}
     ...
     return e
end
]]></code>
```

This form takes two parameters:

- `data` - When the Python window processes event blocks, the `data` parameter specifies an array of Python objects. Each object represents an event in the block. Otherwise, the `data` parameter specifies a single Python object that represents a single event.

  Events include fields that are specified within the `use` element or all fields when the `use` element is not specified.

- `context` - includes variables, data, and resources that are available for the function to use. In the Python window, you can specify the following context:
  - `window` - the window ID of the containing Python window.
  - `input` - the name of the input window.
  - `counter` - the event count of the Python window.
  - `opcode` - opcode of the event
  - `flags` - flag of the event

The second form of an event creation function takes an arbitrary number of parameters as specified in the `use` element when its `expand` attribute is set to **`true`**.

```
<use expand="true">id,value</use>
<code><![CDATA[
 def createEvent(id,value):
      newevent = {}
      newevent["id"] = id
      newevent["value"] = value * 10
      return newevent
 end
]]></code>
```

Both forms of the event creation function can generate zero or more events. When no event is generated, the function should return **`None`**. To generate a single event, the function should return a single Python object. To generate multiple events, an array of Python objects should be returned by the function.

Note: The `expand` attribute of the `use` element is supported only when `process-blocks` is false. When you are processing multiple events, the event creation function must use the first form:

```
def createEvent(data,context):
    ...
end
```

Generating Events

Generating a Single Event

Suppose that you want to generate a single output event per input event. Consider the following Source window:

```
<window-source name="source" insert-only="true">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="string" type="string"/>
      <field name="number" type="int32"/>
      <field name="array" type="array(i32)"/>
    </fields>
  </schema>
</window-source>
```

With an edge between the Source window and the following Python window, the following Python window performs these tasks:

- Copies the `id` from the input event to the output event.
- Encloses the `string` field in double quotation marks.
- Multiplies the value of `number` by 3.
- Multiplies each value in the `array` by 2.

```
<window-python name="python" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="new_string" type="string"/>
      <field name="new_number" type="int32"/>
      <field name="new_array" type="array(i32)"/>
    </fields>
  </schema>
  <copy>id</copy> <!--1 -->
  <code><![CDATA[
def  create(data,context):                          # 2
    e = {}                                          # 3
    e["new_string"] = "\"" + data["id"] + "\""      # 4
```

```
    e["new_number"] = data["number"] * 3          # 5
    e["new_array"] = []                           # 6
    for value in data.array:
        e["new_array"].append(value * 2)
    return e                                      # 7
  ]]></code>
</window-python>
```

1 The `copy` element specified that the `id` is copied from the input event to the output event.

2 Defines a function named `create` that takes two parameters, `data` and `context`.

3 Initializes an empty dictionary named `e`.

4 The `new_string` key is added to the dictionary. The field that is generated encloses the input field string in double quotation marks. Note that the Python concatenation operator is `'+'`.

5 The `new_number` key is added to the dictionary. The field is generated by multiplying the input field number by 3.

6 The `new_array` key is added to the dictionary. The field creates an array (a Python object).

7 Return the Python object created.

When the creation function returns **`None`**, no events are created.

For example, the following code does not create events when the input number field is less than 10:

```
def create(data,context):
     if (data["number"] < 10):
         return None
    ...
end
```

This input event creates the output event that follows;

```
i,n,1,string data,33,[11;24;55]
```

The output event looks like this:

```
<event opcode="insert" window="p/cq/python">
  <value name="id">1</value>
  <value name="new_array">[22;48;110]</value>
  <value name="new_number">99</value>
  <value name="new_string">"string data"</value>
</event>
```

Generating Multiple Events

To generate multiple events, create and return an indexed Python object. The indexed object contains zero or more associative Python objects that represent output events.

For example, consider the following code:

```
def create(data,context):
     events = []

     for i in range(1,5):
         event = []
         -- Add event data here
         events.append(event)
     return events
```

The following Python window uses this code to generate an event for each entry in the array:

```
<window-python name="python" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="new_string" type="string"/>
      <field name="new_number" type="int32"/>
      <field name="array_entry" type="int32"/> <!-- 1 -->
    </fields>
  </schema>
  <code><![CDATA[
        eventId = 1

        def create(data,context):
            global  eventId

            events = []

            for value in data["array"]: # 2
                e = {}

                e["id"] = str(eventId)
                eventId += 1 # 3

                e["new_string"] = "\"" + data["string"] + "\""
                e["new_number"] = data["number"] * 3
                e["array_entry"] = value * 2

                events.append(e)

            return events
  ]]></code>
</window-python>
```

1 This line results in an output event with single field entries instead of an array.

2 These lines create an indexed array and then iterate over the input array field, which generate individual events for each entry. To generate a unique ID for each event, create a local variable named `eventId` and use it to set the output event ID, incrementing it each time after you use it.

3 The `eventID` variable maintains its value between calls into the Python code so that you have a unique, incrementing event ID on your output events. Therefore, you can use the variable value to maintain state in the Python window.

Assigning a Different Event Opcode

By default, the Python window assigns the opcode of the incoming event to the opcode of the outgoing event or set of events. Alternatively, you can use the `_opcode` field to set a different opcode for the outgoing event or set of events:

```
def create(data,context):
     e = {}
     e["new_string"] = "\"" + data["id"] + "\""
     e["new_number"] = data["number"] * 3
     e["new_array"] = []
     for value in data["array"]:
         e["new_array"].append(value * 2)
    e["_opcode"] = "upsert"
     return e
```

Event Forwarding

To publish events more efficiently, you can use event forwarding, which controls the flow of events through a model. For example, if you have an event block that contains multiple large events, you can set the value of `blocksize` to `1` to break down the block and process events individually.

For example, consider the following code:

```
<forwarding suppress="false"> 1
  <destinations>
      <destination window="sourcewin" contquery="cq_1" blocksize="5"
active="true" /> 2
  </destinations>
</forwarding>
```

1 This line enables event forwarding in a Python window. The `suppress` attribute ensures that events are forwarded and generated in the window.

2 This line specifies the Source window to which you want to forward events, the name of the continuous query that contains the Source window, and the number of events to put into each event block. This line also enables events to be forwarded to the destination.

For an example of a complete project that uses event forwarding with a Python window, install the Forwarding example. For more information, see "Install Example Projects" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

4

Utilities

Using Notification Windows

Overview to Notification Windows

Use a Notification window to send notifications through the following delivery channels:

- email using the Simple Mail Transfer Protocol (SMTP)
- text using the Short Message Service (SMS)
- multimedia messages using the Multimedia Messaging Service (MMS)

Notification windows, like Functional windows, enable you to define a function-context to transform incoming events before processing them for possible notifications.

Note: The Notification window uses the SMTP protocol to deliver content for all delivery channels.

Each of the different types of notification has its own configuration requirements. For example, an email message requires that the configuration specify the event field that contains the 'send to' email address. SMS and MMS require phone numbers and phone provider gateway information.

You can format notifications as you want and include the event values within the message. To include event values, include the name of the field, preceded by a **$** character, in your message formatting:

```
<b>$broker</b> sold $quant1 shares of <b>$symbol</b>
$tstap1 for self for $$price1, then sold $quant2 shares for customer
at $tstamp2.
```

Notification windows enable you to create any number of active delivery channels to send notifications. As noted earlier, you can specify functions to determine whether to send the notification. Given the potentially massive amounts of streaming data that could cause an avalanche of notifications, you can specify a `throttle-interval` for each channel. If you set the interval to **`'1 hour'`**, you send at most one notification from that channel to any recipient every hour.

Notification windows never generate events. Nonetheless, you can use the `schema` element to specify values for the `function-context` to generate. You can use these values to format notification messages.

For information about the XML elements associated with a Notification window, see "window-notification" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* and "XML Language Elements Relevant to Notification Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Notification Window Delivery Channels

Specifying the SMTP Server

To use delivery channels, you must specify an `smtp` element to provide information about an SMTP server:

```
<smtp host='host' user='user' password='password' port='port' (opt,
default='25') />
```

Only the `host` attribute of the element is required, because many SMTP servers run on the default port and do not require authentication:

```
<smtp host='mailhost.fyi.sas.com' />
```

However, it is a good practice to supply values for all the attributes of the `smtp` element:

```
<smtp host='smtp-server.host.com'
            user='esptest@hostname.com'
            password='esptest1' port='587' />
```

After you specify information about the SMTP server, you can send notifications through the following delivery channels:

- `email` sends a multipart email message that contains text, HTML, and images to a specified email address
- `sms` sends an SMS text message that contains text to an address in the format ***phoneNumber@gateway***
- `mms` sends an MMS message that contains text and images to an address in the format ***phoneNumber@gateway***

Encrypting Connections

The Notification window encrypts SMTP connections by using one of two protocols: Secure Sockets Layer (SSL) or Transport Layer Security (TLS). SSL has been deprecated, but it is still widely used.

Whenever an email is sent, the client initiates a handshake with the server. The client shares which SSL or TLS versions it is compatible with. The server responds with its digital certificate to confirm its identity. When the certificate is confirmed, the two sides generate and exchange a unique key that is used to decrypt messages between them.

First, a connection needs to be established between the client and the server. By default, an SMTP connection is not secured, which makes it vulnerable to attacks. Therefore, both sides try to establish a secure connection. There are two approaches:

- With explicit security, a client initiates the handshake, and then attempts to upgrade the connection to a secure one by using TLS. When a server's TLS certificate is compatible and no errors occur, a secured connection is established. If anything fails during the process, then a plain-text connection is established.
- With implicit security, a client initiates the handshake without asking a server what encryption method it uses. If the connection is successful, then an SSL connection is set up. When a server's SSL certificate is not compatible or the connection times out, transmission is abandoned.

If the SMTP server that the window connects to is using implicit security, then specify the <ssl> element within the <smtp> element of the window. If the SMTP server is using explicit security, then specify the <tls> element within the `<smtp>` element.

Using the Email Delivery Channel

Here is XML code to use the `email` delivery channel:

```
<email throttle-interval='' test='true | false'>
   <deliver>[code]</deliver>
   <email-info>
      <sender>[sender]</sender>
      <recipients>[recipients]</recipients>
      <subject>[subject]</subject>
      <from>[from]</from>
      <to>[to]</to>
   </email-info>
   <email-contents>
      <text-content name=''>...</text-content>
      <html-content name=''>...</html-content>
      <image-content name=''>...</image-content>
      ...
   </email-contents>
</email>
```

The `email` element contains the following attributes:

- `throttle-interval` specifies a time period in which no more than one notification is sent to a recipient.

- `test` is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This feature can be useful when drafting notification messages.

The `deliver` element is optional. It contains a function to run in order to determine whether the notification should be sent.

The `email-info` element contains functions or hardcoded values that represent the data to be used to send an email notification. It contains the following elements:

- the `sender` email address.
- the `recipients` to whom the email message is sent. Multiple recipients can be delimited by the ',' character or the ';' character.
- the `subject` of the email.
- the `from` text of the email message.
- the `to` text of the email.
- The `email-contents` element, which contains the following elements:
  - the `text-content` element encloses the plain text content of the message.
  - the `html-content` element encloses the HTML content of the message.
  - the `image-content` element encloses a URI to image data. This URI can point to an external entity as follows: **`file:///`*`directory`*`/`*`image_filename`*** or **`http://`*`website`*`/`*`image_filename`***. It can also point to image data directly embedded from an event that is specified in the schema of the Notification window or the immediately preceding window as follows: **`field://`*`fieldname`***.

These elements can be interspersed in any way you want. The content of each element is included in the message in the order in which it appears. Any image data is retrieved and Base64-encoded before being inserted into the message.

Using the SMS Delivery Channel

Here is XML code to use the `sms` delivery channel:

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

The `sms` element contains the following attributes:

- `throttle-interval` specifies a time period in which no more than one notification is sent to a recipient.
- `test` is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This feature can be useful when drafting notification messages.

The `deliver` element is optional. It contains a function to run in order to determine whether the notification should be sent.

The `sms-info` element contains functions or hardcoded values that represent the data to be used to send an SMS text notification. It contains the following elements:

- the `sender` SMS address.
- the `subject` of the message.
- the `from` text of the message.
- the `gateway` element specifies the recipient's provider's SMS gateway. For example, AT&T is `txt.att.net`. Sprint is `messaging.sprintpcs.com`.
- the `sms-contents` element contains the body of the message to be sent. It contains the following element:
  - the `text-content` element encloses the plain text content of the message.

Using the MMS Delivery Channel

Here is XML code to use the `MMS` delivery channel:

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

The `mms` element contains the following attributes:

- `throttle-interval` specifies a time period in which no more than one notification is sent to a recipient.
- `test` is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This feature can be useful when drafting notification messages.

The `deliver` element is optional. It contains a function to run in order to determine whether the notification should be sent.

The `mms-info` element contains functions or hardcoded values that represent the data to be used to send an MMS notification. It contains the following elements:

- the `sender` MMS address.
- the `subject` of the message.
- the `gateway` element specifies the recipient's provider's SMS gateway. For example, AT&T is `txt.att.net`. Sprint is `messaging.sprintpcs.com`.
- the recipient `phone` number.
- the `mms-contents` element contains the body of the message to be sent. It contains the following elements:
  - the `text-content` element encloses the plain text content of the message.
  - the `image-content` element encloses a URI to image data. This URI can point to an external entity as follows: **`file:///`*`directory/image_filename`*** or **`http://`*`website/image_filename`***. It can also point to image data directly embedded from an event that is specified in the schema of the Notification window or the immediately preceding window as follows: **`field://`*`fieldname`***.

These elements can be interspersed in any way you want. The content of each element is included in the message in the order it appears. Any image data is retrieved and Base64-encoded before being inserted into the message.

Using the Function-Context Element

The `function-context` element enables you to define functions to manipulate event data. You can use regular expressions, XML and XPath, or JSON to transform data from complex input information into more usable data.

Here is XML code that uses the function-context element:

```
<function-context>
   <expressions>
      <expression name=''>[Regular Expression]</expression>
      ...
   </expressions>
   <properties>
      <property-map name='' outer='' inner=''>[code]</property-map>
      <property-xml name=''>[code]</property-xml>
      <property-json name=''>[code]</property-json>
      <property-string name=''>[code]</property-string>
      <property-list name='' delimiter=''>[code]</property-list>
      <property-set name='' delimiter=''>[code]</property-set>
      ...
   </properties>
   <functions>
      <function name=''>[code]</function>
      ...
   </functions>
</function-context>
```

You can use two types of functions in the function-context element:

- general functions (for example, `abs`, `ifNext`, and so on)
- functions that are specific to event stream processing (for example, `eventNumber`)

You can reference event fields in either the input event or the output event using the **$** notation (for example, `$[name_of_field]`).

Suppose that you have a name field in the input event and you want to generate an occupation field in the output event based on the value of name. In this case, you could use the following function:

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

Now suppose that you want to add an `hourlyWage` to the output event that depends on occupation:

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

Note: Sequence is important when you define functions in the function-context element. When a function references an output event field, that field needs to be computed before the referring field.

Use POSIX regular expressions in your code. Several functions are available to deal with regular expressions. Because regular expressions must be compiled before they can be used, use the `expressions` element to specify that expressions are compiled a single time when the function context is created. Then, the expression can be referenced from within functions using the following notation:

```
#[name_of_expression]
<function name='myData'>rgx(#myExpression,$inputField,1)
</function>
```

For example, suppose you receive a data field that contains a URI and you want to extract the protocol from it. When you use the following function, the regular expression is compiled each time that the function runs:

```
<function name='protocol'>rgx('(.*):',$uri,1)
</function>
```

If you use the following code, the expression is compiled a single time and used each time that the function runs:

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

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

Reference properties from within functions using the **#** notation: `#[name_of_property]`.

The `properties` element is a container for the following elements:

Element	Description
`property-map`	executes the function to generate a map of name-value pairs to be used for value lookups by name
`property-list`	executes the function to generate a list of strings to be used for indexed access
`property-set`	executes the function to generate a set of strings to be used for value lookups
`property-xml`	executes the function to generate an XML object that can be used for XPath queries
`property-json`	executes the function to generate a JSON object that can be used for JSON lookups
`property-string`	executes the function to generate a string for general use in functions

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

Suppose you had employee information streaming into the model.

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

You can use the `property-map` element to store employee data and examine the `position` field of the event in order to create a `property-xml` that contains the appropriate data. When the employee is a developer, the XML is created from `developerInfo`. Otherwise, it uses `managerInfo`.

Specify the `function-context` element as follows:

```
<function-context>
```

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

When you stream the following events:

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

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

The `function-context` yields the following:

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

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

For more information about how to define each property, see "XML Language Elements for Expressions and Functions" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Using Pattern Windows

Overview of Pattern Windows

Use a Pattern window to detect events of interest (EOIs) as they stream through a project and generate output events as a result of applying pattern logic.

There are three ways to specify an EOI:

- Write a Lua function within a `<code>` element.
- Write a Python function within a `<code>` element.
- Use Expression Engine Language (EEL) within an `<event>` element.

In each case, assemble EOIs into an expression within the `<logic>` element by using operators to apply pattern logic. The expression can also include temporal conditions.

When you use Lua or Python to specify an EOI, you use another Lua or Python function to generate output events. When you use EEL to specify an EOI, you use an `<output>` element to generate output events.

Note: It is recommended to use Lua or Python in Pattern windows. The use of EEL in Pattern windows is legacy functionality.

For information about the XML elements that are associated with a Pattern window, see "window-pattern" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* and "XML Language Elements Relevant to Pattern Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Applying Pattern Logic

Within the `<logic>` element, use the following logical operators to events as operands in order to create pattern logic.

Table 4.1 *Valid Logical Operators for Pattern Logic*

Logical Operator	Function
`and`	All of its operands are true. Takes any number of operands.

Logical Operator	Function
`or`	Any of its operands are true. Takes any number of operands.
`not`	The operand is not true. Takes one operand.
`is`	Ensures that the event that follows the operand is as specified.
`fby`	Each operand is followed by the one after it. Takes any number of operands.
`notoccur`	The operand never occurs. Takes one operand.

The FBY operator (`fby`) is sequential in nature. A single event cannot match on both sides. The left side must be the first to match on an event, and then a subsequent event could match on the right side.

The AND and OR operators are not sequential. Any incoming event can match EOIs on either side of the logical operator and the first matching EOI triggers the variable bindings. Take special care in this case, as this is rarely what you intend when you write a pattern.

You can apply temporal conditions to pattern logic. Consider the following expression:

```
fby(e1, e2, e3)
```

To apply a temporal condition to the `fby` operator, append the condition inside braces. For example, suppose you specify the following expression:

```
fby{10 seconds}(event1,event2,event3)
```

Here, `event1` must occur to start the timer. Then `event2`, followed by `event3`, must occur within 10 seconds of `event1`.

Now suppose you specify the following expression:

```
fby{10 seconds}(fby{10 seconds}(event1,event2),event3)
```

Here, the timer starts when `event1` occurs and `event2` follows within 10 seconds. Then, `event3` must occur within 10 seconds for the entire expression to evaluate to true.

Temporal conditions can be driven in real time or can be defined by a date-time or timestamp field. This field appears in the schema that is associated with the window that feeds the Pattern window. When you use a field-based date-time or timestamp, you must ensure that incoming events are in order with respect to it.

It is a good practice to bound pattern expressions by time. For example, consider the following expression:

```
and{2 seconds}(event1,event2,event3)
```

The open pattern completes either after all three events occur or two seconds after the first event occurs. In contrast, consider the following expression:

```
and(event1,event2,event3)
```

In this case, if `event1` and `event2` occur frequently but `event3` occurs rarely, then many open patterns are created. This behavior might cause memory usage to increase for each new `event1` or `event2` received. Ultimately, that increase could consume all available memory. An INFO message on the console tracks the number of open patterns.

```
[Pattern0025] dfESPpattern::trackHWmark(0x00000000033f7200): [limit: 64] has 33
active instances in pattern window <patternWindow_01>
```

As the number of open instances increases, model performance can degrade.

```
[Pattern0025] dfESPpattern::trackHWmark(0x00000000033f7200): [limit: 4096] has
2049 active instances in pattern window <patternWindow_01>
```

Using Lua in a Pattern Window

Using Lua Functions within Pattern Windows

You can write Lua functions to specify EOIs and to generate output events as the result of applying pattern logic. When you use Lua, complete these actions:

- Use a `<code>` element to contain the following functions:
  - Event of Interest (EOI) functions that return true or false in order to determine whether an event is an EOI
  - An output function that generates the output event when applied pattern-matching criteria have been met
- Enclose events with the `<lua-events>` element. The `<event>` elements refer to Lua functions rather than explicitly specifying events.

For information about additional functionality available when you use Lua in a Pattern window, see Chapter 8, “Common Lua Functionality”.

Writing EOI Functions in Lua

EOI functions in Lua take the following parameters:

- `event` specifies a Lua table that represents the current event in the stream.

  For example:

  ```
  symbol=IBM
  quantity=500
  price=97.34
  ```

```
id=0
broker=John
```

- `context` specifies a Lua table that contains the following information:
  - `window` - the name of the Pattern window

    For example:

    ```
    <window-pattern name='pattern_win'>
    ```

  - `name` - the name of the invoking event

    For example:

    ```
    <lua-events>
         <event name='e1' func='f1'/>
     </lua-events>
    ```

  - `input` - the name of the Source window for incoming events
  - any contextual data that has been returned from other functions
  - `events` - when applicable, all preceding events in the pattern

  For example, consider these events:

```
events=
     paul=
          price=57.34
          symbol=AMZN
          broker=Paul
          id=1
          quantity=500
     john=
          price=97.34
          symbol=IBM
          broker=John
          id=0 quantity=500
     name=george
     window=2
     data=
          mydata=foo
```

The EOI functions should return, at minimum, true or false.

```
function f1(event,context)
    local code = false
    if (event.broker == "John")
    then
        code = true
    end
    return code
end
```

You can also use these functions to put information into the context to be used for subsequent EOI functions. For example, suppose that you want to store the trade price in this context:

```
function f1(event,context)
    local code = false
    if (event.broker == "John")
    then
```

```
            return true, {tradeprice=event.price}
        end
        return false
    end
```

EOI functions can return a Lua table as the second return value. In this case, the table is stored in the context and is available for subsequent EOI functions. In the previous example, you would access the trade price like this:

```
function lowerprice(event,context)
     return(event.price < context.data.tradeprice)
end
```

Writing Output Functions in Lua

The output function is invoked when all relevant EOI functions have been run and a set of events matches the pattern logic. At this point, the ESP server calls the output function and passes it to the pattern context.

```
 data=
     mydata=foo
     window=1
     events= paul=
                 id=1
                 price=57.34
                 quantity=500
                 symbol=AMZN
                 broker=Paul
             john=
                 id=0
                 price=97.34
                 quantity=500
                 symbol=IBM
                 broker=John
```

The context contains any contextual data that has been previously returned from the EOI function as well as all matching events. Events are keyed by event name. You can access event fields as shown here:

```
function output(context)
      local event = {}
      event.symbol_1 = context.events.john.symbol
      event.symbol_2 = context.events.paul.symbol
      return event
end
```

The returned event is assigned a unique ID and injected into the event stream.

Defining Events

In a Lua-based Pattern window, `<event>` elements refer to EOI functions written in Lua instead of containing expression code. In the following code snippet, the event

john refers to an EOI function named f1, and the event paul refers to an EOI function named f2.

```
<lua-events>
    <event name="john" func="f1" />
    <event name="paul" func="f2" />
</lua-events>
```

By default, all event fields are passed from the ESP server into the EOI function. To maximize efficiency, you can specify that you want to send only certain event fields to the function. For example, the following definition tells the ESP server to send only two fields, broker and price, into the EOI function f1.

```
<lua-events>
     <event name="e1" func="f1">
       <use>broker,price</use>
     </event>
</lua-events>
```

This action can be effective when you have dozens of event fields but use only a small set of them in your EOI function.

When applicable, you can send the entire set of previously matched events into a function with the sendevents attribute of the <event> element.

```
<lua-events>
      <event name="paul" func="f2" sendevents="true" />
</lua-events>
```

When you specify sendevents, the ESP server puts all of the previously matched events into the context in an events field:

```
name=paul
events=
    john=
        broker=John
        price=97.34
        quantity=500
        symbol=IBM
        id=0
```

These events are referenced by the event name that they are associated with (the event names that they matched). This makes it easy to compare data between the events in the entire stack.

Note: When you use a small set of fields, it is more efficient to use the context data than to put all the events onto the stack each time.

Repeating Events

Many patterns look for an EOI that repeats. The Lua-based Pattern window provides a simple mechanism to use event definitions and pattern logic to detect repetition. Simply add *:number* after the element that you want to repeat. For example, suppose that you wanted to look for an event e1 that repeats five times.

```
fby(e1:5)
```

You can add the number to either an operator or to an event.

```
fby(is:5(e1))
```

Suppose that you wanted to capture three successive stock trades where each price is lower than the previous (without an intervening trade where the price was higher). You could use the following code:

```
<pattern index="symbol">
   <lua-events>
      <event name="start" func="start" /> <!-- 1 -->
      <event name="decrease" func="f1" /> <!-- 2 -->
   </lua-events>
   <logic>fby(start,is:2(decrease))</logic> <!-- 3 -->
   <code output="output">
      <![CDATA[
       function start(event,context)
           return true, {price=event.price}
       end

       function f1(event,context)
           if (event.price < context.data.price)
           then
               return true, {price=event.price}
           end

           return false
       end ...
</pattern>
```

1 Define a start event that returns true for every event and sets the trade price in the context.

2 Define a decrease event that returns true if the current trade price is less than the previous trade price (and also sets the trade price in the context).

3 Use the repeating logic syntax to require that the decrease event happen twice.

When you have repeating events, any EOI that occurs within the repeating sequence is represented within the context data as an array. So for the preceding example, the output function can access `decrease` events as follows:

```
function output(context)
   local event = {}
   -- Loop through all decrease events
   for index,value in ipairs(context.events.decrease) do
   -- set event values
   end
   return event
end
```

This functionality enables you to change the number of repetitions that you are looking for without revising the code. In the previous example, you can increase the pattern to look for four `decreasing` price trades by changing the repeat modifier in the `logic` tag as follows:

```
<logic>fby(start,is:3(decrease))</logic>
```

Using Arrays

You can use Lua code within the Pattern window to write event array fields. This capability can be useful when you need to draw from the same fields from several EOIs for output.

Suppose that you have a pattern that finds five stock trades: :

```
<field name="price_1" type="double"/>
<field name="price_2" type="double"/>
<field name="price_3" type="double"/>
<field name="price_4" type="double"/>
<field name="price_5" type="double"/>
```

Instead of a list of enumerated price fields, you can specify a price array:

```
<field name="prices" type="array(dbl)"/>
```

After you assign values to elements of the array in the Lua code, you can process them as follows:

```
...
local event = {}
event.prices[1] = event1.price
event.prices[2] = event2.price
event.prices[3] = event3.price
event.prices[4] = event4.price
event.prices[5] = event5.price
...
```

Using Python in a Pattern Window

Using Python Functions within Pattern Windows

You can write Python functions to specify EOI (Event of Interest) functions and to generate output events as the result of applying pattern logic. When you use Python, complete these actions:

- Use a `<code>` element to contain the following functions:
  - An EOI function that returns true or false in order to determine whether an event is an EOI
  - An output function that generates the output event when applied pattern-matching criteria have been met
  - An optional initialization function to call before the model starts.
  - An optional destroy function to call when the window is destroyed.

- Enclose events with the <python-events> element. The <event> elements refer to Python functions rather than explicitly specifying events.

For information about additional functionality that is available when you use Python in a Pattern window, see Chapter 9, "Common Python Functionality".

Writing EOI Functions in Python

EOI functions in Python take the following parameters:

- event specifies a Python object that represents the current event in the stream. For example:

```
{'id': 0, 'broker': 'John', 'symbol': 'IBM', 'price': 97.34,
'quantity': 500}
```

- context includes variables, data, and resrouces that are available for the EOI function to use. In a Pattern window, context can contain the following information:
  - window - the name of the Pattern window

    For example:

    ```
    <window-pattern name='pattern_win' >
    ```

  - name - the name of the invoking event

    For example:

    ```
    <python-events>
         <event name='e1' func='f1'/>
     </python-events>
    ```

  - input - the name of the Source window for incoming events
  - any contextual data that has been returned from other functions
  - events - when applicable, all preceding events in the pattern

  For example, consider the following:

```
{
    'name': 'george',
    'input': 's',
    'window': 'pattern',
    'events': {
        'john': {'id': 0, 'broker': 'John', 'symbol': 'IBM', 'price':
97.34, 'quantity': 500},
        'paul': {'id': 3, 'broker': 'Paul', 'symbol': 'IBM', 'price':
57.34, 'quantity': 500}
    }
}
```

The EOI functions should return, at minimum, true or false.

```
def f1(event,context):
    code = False

    if (event["broker"] == "John"):
        code = True
```

```
    return code
```

You can also use these functions to put information into the context to be used for subsequent EOI functions. For example, suppose that you want to store the trade price in this context:

```
def f1(event,context):
    code = False

    if (event["broker"] == "John"):
        return True, {"tradeprice":event["price"]}

    return False
```

EOI functions can return a Python object as the second return value. In this case, the object is stored in the context and is available for subsequent EOI functions. In the previous example, you would access the trade price like this:

```
def lowerprice(event,context):
    return(event["price"] < context["data"]["tradeprice"])
```

Writing Output Functions in Python

The output function is invoked when all relevant EOI functions have been run and a set of events matches the pattern logic. At this point, the ESP server calls the output function and passes it to the pattern context.

The context contains any contextual data that has been previously returned from the EOI function as well as from all matching events. Events are keyed by event name. You can access event fields as shown here:

```
def output(context):
    event = {}
    event["symbol_1"] = context["events"]["john"]["symbol"]
    event["symbol_2"] = context["events"]["paul"]["symbol"]
    return event
```

The returned event is assigned a unique ID and injected into the event stream.

Defining Events

In a Python-based Pattern window, `<event>` elements refer to EOI functions that are written in Python instead of functions that contain expression code. In the following code snippet, the event `john` refers to an EOI function named `f1`, and the event `paul` refers to an EOI function named `f2`.

```
<python-events>
    <event name="john" func="f1" />
    <event name="paul" func="f2" />
</python-events>
```

By default, all event fields are passed from the ESP server into the EOI function. To maximize efficiency, you can specify that you want to send only certain event fields

to the function. For example, the following definition tells the ESP server to send only two fields, `broker` and `price`, into the EOI function `f1`.

```
<python-events>
     <event name="e1" func="f1">
       <use>broker,price</use>
     </event>
</python-events>
```

This action can be effective when you have dozens of event fields but use only a small set of them in your EOI function.

When applicable, you can send the entire set of previously matched events into a function with the `sendevents` attribute of the `<event>` element.

```
<python-events>
      <event name="paul" func="f2" sendevents="true" />
</python-events>
```

When you specify `sendevents`, the ESP server puts all of the previously matched events into the context in an events field:

```
{
    'name': 'george',
    'input': 's',
    'window': 'pattern',
    'events': {
        'john': {'id': 0, 'broker': 'John', 'symbol': 'IBM', 'price':
97.34, 'quantity': 500},
        'paul': {'id': 3, 'broker': 'Paul', 'symbol': 'IBM', 'price':
57.34, 'quantity': 500}
    }
}
```

These events are referenced by the event name that they are associated with (the event names that they matched). This makes it easy to compare data between the events in the entire stack.

Note: When you use a small set of fields, it is more efficient to use the context data than to put all the events onto the stack each time.

Repeating Events

Many patterns look for an EOI function that repeats. The Python-based Pattern window provides a simple mechanism to use event definitions and pattern logic to detect repetition. Simply add `:`*`number`* after the element that you want to repeat. For example, suppose that you wanted to look for an event `e1` that repeats five times.

```
fby(e1:5)
```

You can add the number to either an operator or to an event.

```
fby(is:5(e1))
```

Suppose that you wanted to capture three successive stock trades where each price is lower than the previous (without an intervening trade where the price was higher). You could use the following code:

```
<pattern index="symbol">
   <python-events>
      <event name="start" func="start" /> <!-- 1 -->
      <event name="decrease" func="f1" /> <!-- 2 -->
   </python-events>
   <logic>fby(start,is:2(decrease))</logic> <!-- 3 -->
   <code output="output">
      <![CDATA[
    def start(event,context):
        return True, {"price":event["price"]}

    def f1(event,context):
        if (event["price"] < context["data"]["price"]):
           return True, {"price":event["price"]}
        ...
</pattern>
```

1 Define a start event that returns true for every event and sets the trade price in the context.

2 Define a decrease event that returns true if the current trade price is less than the previous trade price (and also sets the trade price in the context).

3 Use the repeating logic syntax to require that the decrease event happen twice.

When you have repeating events, any EOI function that occurs within the repeating sequence is represented within the context data as an array. So for the preceding example, the output function can access `decrease` events as follows:

```
def output(context):
    event = {}

    -- Loop through all decrease events

    for value in context["events"]["decrease"]::
        -- set event values

    return event
```

This functionality enables you to change the number of repetitions that you are looking for without revising the code. In the previous example, you can increase the pattern to look for four `decreasing` price trades by changing the repeat modifier in the `logic` tag as follows:

```
<logic>fby(start,is:3(decrease))</logic>
```

Using Arrays

You can use Python code within the Pattern window to write event array fields. This capability can be useful when you need to draw from the same fields from several EOI functions for output.

Suppose that you have a pattern that finds five stock trades:

```
<field name="price_1" type="double"/>
<field name="price_2" type="double"/>
<field name="price_3" type="double"/>
<field name="price_4" type="double"/>
<field name="price_5" type="double"/>
```

Instead of a list of enumerated price fields, you can specify a price array:

```
<field name="prices" type="array(dbl)"/>
```

After you assign values to elements of the array in the Python code, you can process them as follows:

```
...
event = {}
event["prices"] = []
event["prices"].append(event1["price"])
event["prices"].append(event2["price"])
event["prices"].append(event3["price"])
event["prices"].append(event4["price"])
event["prices"].append(event5["price"])
...
```

Using Expression Engine Language

When using Expression Engine Language (EEL), specify EOIs within an `<event>` element by providing the following information:

- a pointer for the window where the event is coming from.
- a string name for the EOI.
- a WHERE clause on the fields of the incoming event. The WHERE clause can include a number of unification variables (bindings).

For example, in the following list of EOIs, you can see this information:

- The pointer is to `src_win`.
- The names of the EOIs are `e1`, `e2`, and `e3`.
- The WHERE clause includes the unification variables `symbol`, `price`, and `b`.

```
<event source="src_win" name="e1">symbol=="IBM" and price > 101.00 and
b == buy</event>
<event source="src_win" name="e2">symbol=="T" and price > 30.000 and b
== buy</event>
<event source="src_win" name="e3">symbol=="AMZN" and price > 1800.00
and b == buy</event>
```

For more information about Expression Engine Language (EEL), see *Expression Language: Reference Guide*.

Here is an XML example of a pattern from a broker surveillance model.

```
<pattern>
        <events>
```

```
            <event name='e1'>((buysellflg==1) and (broker == buyer)
            and (s == symbol) and (b == broker) and (p == price))</
event>
            <event name='e2'>((buysellflg==1) and (broker != buyer)
            and (s == symbol) and (b == broker))</event>
            <event name='e3'><![CDATA[((buysellflg==0) and (broker ==
seller)
            and (s == symbol) and (b == broker) and (p < price))]]></
event>
        </events>
        <logic>fby{1 hour}(fby{1 hour}(e1,e2),e3)</logic>
        ...
    </pattern>
    <pattern>
        <events>
            <event name='e1'>((buysellflg==0) and (broker == seller)
            and (s == symbol) and (b == broker))</event>
            <event name='e2'>((buysellflg==0) and (broker != seller)
            and (s == symbol) and (b == broker))</event>
        </events>
        <logic>fby{10 minutes}(e1,e2)</logic>
        ...
    </pattern>
</patterns>
</window-pattern>
```

Here is an XML example of a pattern from an e-commerce model:

```
<pattern>
        <events>
            <event name='e1'>eventname=='ProductView'
            and c==customer and p==product</event>
            <event name='e2'>eventname=='AddToCart'
            and c==customer and p==product</event>
            <event name='e3'>eventname=='CompletePurchase'
            and c==customer</event>
            <event name='e4'>eventname=='Sessions'
            and c==customer</event>
            <event name='e5'>eventname=='ProductView'
            and c==customer and p!=product</event>
            <event name='e6'>eventname=='EndSession'
            and c==customer</event>
        </events>
        <logic>fby(e1,fby(e2,not(e3)),e4,e5,e6)</logic>
        ...
    </pattern>
```

You can define multiple patterns within a Pattern window. Each pattern typically has multiple EOIs, possibly from multiple windows or just one input window.

Suppose that there is a single window that feeds a Pattern window, and the associated schema is as follows:

```
<schema>
   <fields>
      <field name="ID" type="int32" key="true"/>
      <field name="symbol" type="string"/>
      <field name="price" type="double"/>
```

```
      <field name="buy" type="int32"/>
      <field name="tradeTime" type="date"/>
   </fields>
</schema>
```

Suppose further that there are two EOIs and that their relationship is temporal. You are interested in one event followed by the other within some period of time. This scenario is depicted in the following code segment:

```
<pattern>
   <events>
      <!-- Someone buys IBM at price > 150.00 followed within
           5 seconds of buying MSFT at price > 110.00 -->
      <event name="e1">symbol=="IBM" and price > 150.00 and b==buy</
event>
      <event name="e2">symbol=="MSFT" and price > 110.00 and b==buy</
event>
    </events>
   <logic>fby{5 seconds}(e1, e2)</logic>
...
</pattern>
```

There are two EOIs, `e1` and `e2`. The beginning of the WHERE clauses is standard: `symbol==`*`constant`* `and price>`*`constant`*. The last part of each WHERE clause is where event unification occurs.

Because `b` is not a field in the incoming event, it is a free variable that is bound when an event arrives. It matches the first portion of the WHERE clause for event `e1` (for example, an event for IBM with price > 150.00.) In this case, `b` is set to the value of the field `buy` in the matched event. This value of `b` is then used in evaluating the WHERE clause for subsequent events that are candidates for matching the second event of interest `e2`. The added unification clause `and b == buy` in each event of interest ensures that the same value for the field `buy` appears in both matching events.

For example, suppose that the incoming schema is as defined previously and you define the following pattern:

```
<pattern>
   <events>
      <!-- Someone buys or sells IBM at price > 150.00 and
           buys or sells IBM at a price > 152.00 within 5 seconds -->
      <event name="e1"> symbol=="IBM" and price > 150.00</event>
      <event name="e2"> symbol=="IBM" and price > 152.00</event>
    </events>
   <logic>and{5 seconds}(e1, e2)</logic>
...
</pattern>
```

Suppose an event comes into the window where symbol is "IBM" and price is "152.10". Because this is an AND operator, no inherent sequencing is involved, and the WHERE clause is satisfied for either side of the operator by that event. Thus, the pattern becomes true, and event `e1` is the same as event `e2`. This is probably not what you intended. Therefore, you can make slight changes to the pattern as follows:

```
<pattern>
   <events>
```

```
      <!-- Someone buys or sells IBM at price > 150.00 and <=152.00 and
           buys or sells IBM at a price > 152.00 within 5 seconds -->
      <event name="e1"> symbol=="IBM" and price > 150.00 and price <=
152.00</event>
      <event name="e2"> symbol=="IBM" and price > 152.00)</event>
    </events>
   <logic>and{5 seconds}(e1, e2)</logic>
...
</pattern>
```

After you make these changes, the price clauses in the two WHERE clauses disambiguate the events so that a single event cannot match both sides. It requires two unique events for the pattern match to occur.

Suppose that you specify a temporal condition for an AND operator such that event `e1` and event `e2` must occur within five seconds of one another. In that case, temporal conditions for each of the events are optional.

State Definitions for Operator Trees

An operator tree is a hierarchical structure that defines how streaming data flows through a series of connected operators. Each operator performs a specific task and passes the results downstream, enabling complex event processing through modular building blocks.

Operator trees can have one of the following states:

- initial - no events have been applied to the tree.
- waiting - an event has been applied causing a state change, but the left (and right, if applicable) arguments do not yet permit the tree to evaluate to TRUE or FALSE.
- TRUE or FALSE - sufficient events have been applied for the tree to evaluate to one of these logical Boolean values.

The state value of an operator sub-tree can be FIXED or not-FIXED. When the state value is FIXED, no further events should be applied to it. When the state value is not-FIXED, the state value could change based on application of an event. New events should be applied to the sub-tree.

When a pattern instance fails to emit a match and destroys itself, it folds. The instance is freed and removed from the active pattern instance list. When the top-level tree in a pattern instance (the root node) becomes FALSE, the pattern folds. When it becomes TRUE, the pattern emits a match and destroys itself.

An operator tree (*OPT*) is a tree of operators and EOIs. Given that *EOI* refers to an event of interest or operator tree (*EOI* | *OPT*):

Table 4.2 *Expressions and Associated Outcomes*

Expression	Outcome
`not` *EOI*	A Boolean negation. This expression remains in the waiting state until *EOI* evaluates to TRUE or FALSE. Then it performs the logical negation. It becomes FIXED only when *EOI* becomes FIXED. **Note:** It is recommended to always use `not` with a time limit. Otherwise, you could wait indefinitely for something not to occur.
`not` *OPT*	A Boolean negation. This expression remains in the waiting state until *OPT* evaluates to TRUE or FALSE. Then it performs the logical negation. It becomes FIXED only when *OPT* becomes FIXED. **Note:** It is recommended to always use `not` with a time limit. Otherwise, you could wait indefinitely for something not to occur.
`notoccur` *EOI*	Becomes TRUE on application of an event that does not satisfy the *EOI*, but it is not marked FIXED. This expression implies that more events can be applied to it. As soon as it sees an event that matches the *EOI*, it becomes FALSE and FIXED.
`notoccur` *OPT*	Not permitted.
EO `or` *EO*	An event that is always applied to all non-FIXED sub-trees. This expression becomes TRUE when one of its two sub-trees becomes TRUE. It becomes FALSE when both of the sub-trees become FALSE. It is FIXED when one of its sub-trees is TRUE and FIXED if both of its sub-trees are FALSE and not FIXED.
EO `and` *EO*	An event that is always applied to all non-FIXED sub-trees. This expression becomes TRUE when both of its two sub-trees become TRUE. It becomes FALSE when one of the sub-trees becomes FALSE. It is FIXED when one of its sub-trees is FALSE and FIXED or both of its sub-trees are TRUE and FIXED.
EO `FBY` *EO*	Attempts to complete the left hand side (LHS) with the minimal number of event applications before applying events to the right hand side (RHS). The apply rule is as follows: ■ If the LHS is not TRUE or FALSE, apply the event to the LHS until it becomes TRUE or FALSE. ■ When the LHS becomes FALSE, set the followed by state to FALSE and become FIXED.

Expression	Outcome
	■ When the LHS becomes TRUE, apply all further events to the RHS until the RHS becomes TRUE or FALSE. If the RHS becomes FALSE, set the `FBY` state to FALSE and FIXED. If it becomes TRUE, set the `FBY` state toTRUE and FIXED. This algorithm seeks the minimal length sequence of events that completes an `FBY` pattern.
`is` *EOI*	Becomes TRUE on the application of an event that satisfies the EOI and becomes FALSE otherwise. Becomes FIXED on the first application of an event.
`is` *OPT*	Not permitted.

Table 4.3 *Logic Underlying a Set of Sample Operator Trees and Events*

Logic	Description
`(a fby b)`	Detect `a, ..., b`, where `...` can be any sequence.
`( a fby ( (notoccur c) and b ) )`	Detect `a, ..., b`: but there can be no `c` between `a` and `b`.
`( a fby (not c) ) fby (not (not b))`	Detect `a, X, b`: when `X` cannot be `c`.
`( ( (a fby b) fby (c fby d) ) and (notoccur k) )`	Detect `a, ..., b, ..., c, ..., d` : but `k` does not occur anywhere in the sequence.
`a fby (notoccur(c) and b)`	Detect `a FBY b` with no occurrences of `c` in the sequence.
`is(a) fby is(b)`	Detect `a FBY b` directly, with nothing between `a` and `b`.
`a fby (b fby ((notoccur c) and d))`	Detect `a ... b ... d`, with no occurrences of `c` between `a` and `b`.
`(notoccur c) and (a fby (b fby d))`	Detect `a ... b ... d`, with no occurrences of `c` anywhere.

Restrictions on Patterns

The following restrictions apply to patterns that you define in Pattern windows:

- The data type of the key field must be `int64`.
- An event of interest should be used in only one position of the operator tree. For example, the following code would return an error:

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

- Pattern windows work only on Insert events.

  If there is an input window generating updates or deletions, then you must place a Procedural window between the input window and the Pattern window. The Procedural window then filters out or transforms non-insert data to insert data.

  Patterns also generate only Inserts. The events that are generated by Pattern windows are indications that a pattern has successfully detected the sequence of events that they were defined to detect. The schema of a pattern consists of a monotonically increasing pattern HIT count in addition to the non-key fields that you specify from events of interest in the pattern.

  ```
  dfESPpattern::addOutputField() and dfESPpattern::addOutputExpression()
  ```

- When defining the WHERE clause expression for pattern events of interest, binding variables must always be on the left side of the comparison (for example, `bindvar == field`) and cannot be manipulated.

  For example, in the following C++ code, the `addEvent` statement would be flagged as invalid:

  ```
  e1 = consec->addEvent(readingsWstats, "e1",
       "((vmin < aveVMIN) and (rCNT==MeterReadingCnt) and (mID==meterID))");
  e2 = consec->addEvent(readingsWstats, "e2",
       "((mID==meterID) and (rCNT+1==MeterReadingCnt) and (vmin < aveVMIN))");
  op1 = consec->fby_op(e1, e2,28800000001);
  ```

  Consider the WHERE clause in **`e1`**. It is the first event of interest to match because the operator between these events is a followed-by. It ensures that event field **`vmin`** is less than field **`aveVMIN`**. When this is true, it binds the variable **`rCNT`** to the current meter reading count and binds the variable **`mID`** to the **`meterID`** field.

  Now consider **`e2`**. Ensure that the following criteria are met:

  - The **`meterID`** is the same for both events.
  - The meter readings are consecutive based on the **`meterReadingCnt`**.
  - **`vmin`** for the second event is less than **`aveVMIN`**.

  The error in this expression is that it checked whether the meter readings were consecutive by increasing the **`rCNT`** variable by 1 and comparing that against the current meter reading. Variables cannot be manipulated. Instead, you confine manipulation to the right side of the comparison to keep the variable clean.

The following code shows the correct way to accomplish this check. You want to make sure that meter readings are consecutive (given that you are decrementing the meter reading field of the current event, rather than incrementing the variable).

```
e1 = consec->addEvent(readingsWstats, "e1",
     "((vmin < aveVMIN) and (rCNT==MeterReadingCnt) and (mID==meterID))");
e2 = consec->addEvent(readingsWstats, "e2",
     "((mID==meterID) and (rCNT==MeterReadingCnt-1) and (vmin < aveVMIN))");
op1 = consec->fby_op(e1, e2,2880000000l);
```

Using Stateless Pattern Windows

Pattern windows are insert-only with respect to both their input windows and the output that they produce. The output of a Pattern window is a monotonically increasing integer ID that represents the number of patterns found in the Pattern window. The ID is followed by an arbitrary number of non-key fields that are assembled from the fields of the events of interest for the pattern.

Because both the input and output of a Pattern window are unbounded and insert-only, they are natural candidates for stateless windows (that is, windows with index type `pi_EMPTY`). Usually, you want to have a Copy window with a retention policy follow any insert-only window.

Pattern windows are automatically marked as insert-only. They reject records that are not Inserts. Thus, no problems are encountered when you use an index type of `pi_EMPTY` with Pattern windows. If a Source window feeds the Pattern window, then the Source window needs to be explicitly told that it is insert-only, using the `dfESPwindow::setInsertOnly()` call. This causes the Source window to reject any events with an opcode other than Insert, and permits an index type of `pi_EMPTY` to be used.

Stateless windows are efficient with respect to memory use. More than one billion events have been run through pattern detection scenarios such as this with only modest memory use (less than 500MB total memory).

```
Source Window [insert only, pi_EMPTY index] --> PatternWindow[insert
only,
     pi_EMPTY index]
```

Enabling the Heartbeat Interval

Patterns that can time out are sent heartbeats by the system. When there are millions of open, uncompleted patterns, the default heartbeat interval of one second is too short. In this case, the system attempts to time out every pattern each second, and that can slow system performance.

To remedy this problem, tune the heartbeat interval:

- In XML, use the following attribute on the `project` element: `heartbeat-interval='number-of-seconds'`
- In C++, call `dfESPproject::setHeartbeatInterval(int number-of-seconds)` before you start the project.

Set the *number-of-seconds* as high as is practical.

Using Index Generation Functions

An index generation function selects fields in an event to sort them before patterns are applied. For example, consider the following EOIs and pattern logic:

```
<event name='e1'>(sym == symbol) and (price > 100)</event>
<event name='e2'>(sym == symbol) and (price < 100)</event>
<logic>fby(e1,e2)</logic>
```

Here, you are detecting events that have the same symbol and tracking them when the price first exceeds 100, and then falls below 100. Because `sym` is part of the EOI, every event that streams into the Pattern window is checked against every instance of this pattern. When applied to many events, that evaluation could degrade project performance.

Alternatively, you can generate an index of `sym` on the pattern:

```
<window-pattern...index='sym'>
```

In this case, every event is first sorted by `sym` because it is the index. After that, the pattern is applied to events. When `sym='IBM'`, only events that match the index are evaluated. Hence, you can simplify the EOIs:

```
<event name='e1'>price > 100</event>
<event name='e2'>price < 100</event>
```

This process can improve performance because the Pattern window has fewer events to evaluate.

Using Counter Windows

Use a Counter window to determine how many events are streaming through your model and the rate at which they are being processed.

For information about the XML elements associated with Counter windows, see “window-counter” in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

You cannot configure the output schema for a Counter window. It is hardcoded to generate events with the following format:

```
"input*:string,totalCount:int64,totalSeconds:double,totalRate:double,i
ntervalCount:int64,intervalSeconds:double,intervalRate:double" .
```

The value of *input* is the name of the window that sent the event to the Counter window. The remaining fields report performance statistics.

Table 4.4 *Output Schema Fields*

Name	Description
`totalCount`	The number of events that have entered the window since the project started (or since the count was cleared).
`totalSeconds`	The number of seconds that the window has received events since the project started (or since the count was cleared).
`totalRate`	`totalCount` / `totalSeconds`
`intervalCount`	The number of events that have entered the window for the specified interval.
`intervalSeconds`	The actual number of seconds for the interval. This value is close to but not exactly the same as the specified interval.
`intervalRate`	`intervalCount` / `intervalSeconds`

When you specify a `count-interval`, the Counter window generates events regularly at that interval that report performance statistics. When the Counter window has not received any events since the last event was generated, it does not generate a new event.

```
<window-counter name='counter'
                        count-interval='2 seconds'
                        clear-interval='30 seconds'/>
```

```
<event opcode='upsert' window='trades/trades/counter'>
      <value name='input'>trades</value>
      <value name='intervalCount'>288215</value>
      <value name='intervalRate'>144108</value>
      <value name='intervalSeconds'>2</value>
      <value name='totalCount'>794312</value>
      <value name='totalRate'>132385</value>
      <value name='totalSeconds'>6</value>
   </event>
```

If you do not specify a `count-interval`, the value defaults to one second and an event that contains only overall values is generated.

```
<window-counter name='counter'/>
```

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

The opcode for generated events is based on the index of the Counter window. If the index is `pi_EMPTY`, the opcode is Insert. For any other index value, the opcode is Upsert.

To use a Counter window, add an edge with the Counter window as the target and the window to monitor as the source. You can connect multiple windows to the same Counter window. To show the results, you can add the Counter window to the trace attribute of the `<contquery>` element that prints formatted events to standard output.

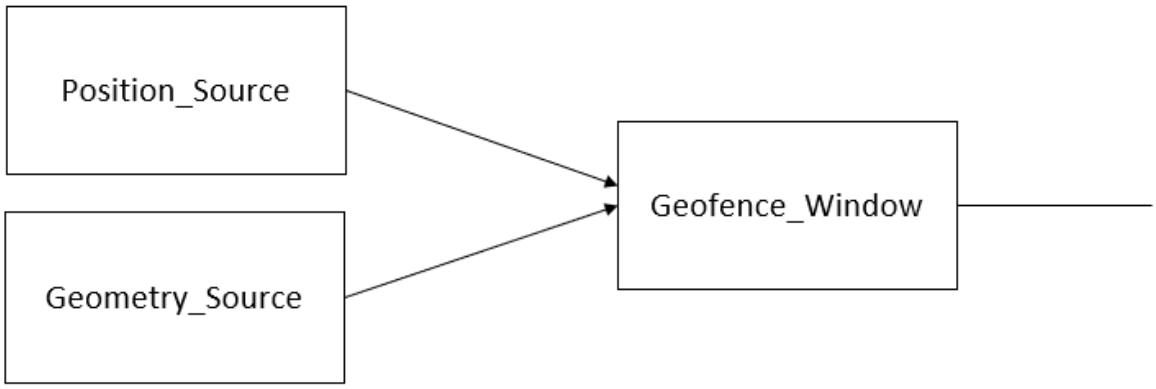
Using Geofence Windows

Overview to Geofence Windows

A *geofence* is a virtual perimeter for a real-world geographic area. You can dynamically generate a geofence as a radius around a specific location or create one as a set of specific boundaries. The Geofence window determines whether the location of an event stream is inside or close to an area of interest. You can augment an event with location details.

Geofence windows require two input windows: one to inject streaming events and another to inject geofence areas and locations. By default, you connect the window that injects events to the Geofence window with the first edge that you specify in the project. You connect the window that injects geofence areas and locations to the Geofence window with the second edge.

```
<edges>
   <edge source='position_source' target='geofence_window'/>
   <edge source='geometry_source' target='geofence_window'/>
</edges>
```



Thus, the Geofence window behaves like an outer Join window or a lookup operation. The events are on the streaming side and the geofence areas and locations are on the dimension side.

Alternatively, you can explicitly assign a role to the edges that connect input windows to Geofence Windows: **`position`** or **`geometry`**. For example:

```
<edges>
   <edge source='geometry_source' target='geofence_window'
role='geometry'/>
   <edge source='position_source' target='geofence_window'
role='position'/>
</edges>
```

The Geofence window is designed to support Cartesian or geographic coordinate types. The only requirement is that all coordinates must be consistent and must refer to the same space or projection. For geographic coordinates, the coordinates must be specified in the (X,Y) Cartesian order (longitude, latitude). All distances are defined and calculated in meters.

You can restrict the geofence lookup to selected geometries depending on input position metadata. For example, if your project receives events that identify the position of multiple automobiles, you can choose to look up only geometry geofences relevant to a specific subset. You do this using the `join-key-fieldname` attributes of the `geometry` and `position` inputs. Attribute values are compared and evaluated before a geofence lookup is performed.

For information about the XML elements associated with Geofence windows, see "window-geofence" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* and "XML Language Elements Relevant to Geofence Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

You can use the following ESP server properties to specify a localized value for a geometry type in the output map of a Geofence window.

Table 4.5 *Geofence Properties*

Property	Default Value
`Geofence0042`	**`circle`**

Property	Default Value
`Geofence0043`	**`polygon`**
`Geofence0044`	**`polyline`**
`Geofence0045`	**`unknown`**

Geometries

Overview

Areas and locations of interest are defined as *geometries*. The Geofence window supports the following geometries: polygons, polylines, and circles. Geometries are published as events, one event per geometry.

Use the `geotype-fieldname` attribute of the `geometry` element to specify what type of geometry to include. When this attribute is not specified, the Geofence window determines the type automatically, based on the data characteristics. When the data does not match the type specified (for example, data specifies a non-closed ring but `geotype-fieldname='polygon'`), the geometry is rejected.

The Geofence window supports Insert, Update, and Delete opcodes, which can dynamically update the geometries. Each Geofence window instance can implement polygon geometries, polyline geometries, or circle geometries, and it can perform geofence analysis on all types simultaneously.

Remember that the Geofence window behaves like a lookup join. You must build its output schema using all or a subset of the fields that come from the geometries input window. For more information about the output schema, see "Output Schema".

Polygons

A *polygon* is a plane shape representing an area of interest. The Geofence window supports polygons, multi-polygons, and polygons with holes or multiple rings.

Define a polygon as a list of position coordinates that represent the polygon's ring(s). A *ring* is a closed list of position coordinates. To be considered closed, the last point of the ring list must be the same as the first one. For example, a ring that is geometrically defined with four points, like a square, must declare five position coordinates, the last being the same as the first.

The input polygon window schema must have at least the following two fields:

- A single key field of type Int64 or String. This field defines the ID of the polygon.
- A data field of type array(dbl).

You can also specify an optional radius field of type Double. This field represents a proximity distance around the polygon. When this field is not specified, the default distance that is defined by the parameter radius is used. This value is used only when the property proximity-analysis is set to **`true`**.

When the polygon is not closed and does not contains any closed ring, it is considered a polyline.

For example, the following string represents a polygon made of four points. Notice that the fifth pair of numbers is identical to the first.

```
5.281 9.455 3.607 7.112 6.268 6.181 8.414 7.705 5.281 9.455
```

For polygons with multiple rings, the first ring defined must be the exterior ring and any others must be interior rings or holes.

Note: Overlapping holes for a polygon's definition are not supported.

The schema can also have an optional description field. All other fields are ignored.

When working with polygons, the Geofence window analyzes each event position coming from the streaming window and returns the polygon in which this position is inside. If there are multiple matching geometries (in cases of overlapping polygons) and if the option `output-multiple-results` is set to true, then multiple events are produced (one per geometry).

In addition, when the property proximity-analysis is set to **`true`**, the Geofence window returns the ID of the polygon that is within the distance defined by the radius property value. Distance is measured from the position to the closest outer ring segment (holes are ignored).

When a polygon is detected, the output `geodistance-fieldname` returns the following:

- When the `polygon-compute-distance-to` property is set to **`segment`**: the exact distance from the position to the polygon. This value is negative when the position is within the polygon and it is positive when it is outside. When proximity-analysis is set to true, a polygon crossing can then easily be detected by a sign change using a Pattern window.
- When the `polygon-compute-distance-to` property is set to **`centroid`**: the exact distance from the position to the centroid of the polygon.

Polylines

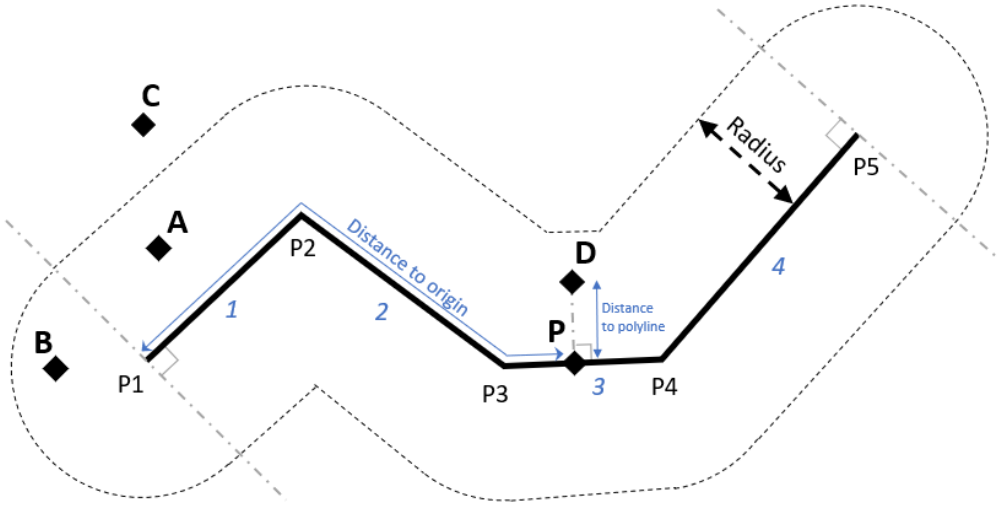
A polyline is a shape that represents a border or a tripwire. Define a polyline as a list of coordinates that represent the polyline's segment or segments. The sequence of the points in the segment definition defines the polyline's vector direction and its left and right side.

When working with polylines, a Geofence window analyzes each event position that comes from the positions Source window. It returns the ID of the polyline that is within the distance defined by the radius property value. Distance is measured from the position to the closest segment.

When a polyline is detected, the output `geodistance-fieldname` returns the exact distance from the event position to the polyline. This value is negative whenever the position is on the left side of the polyline. It is positive whenever it is on the right side. A tripwire crossing can then easily be detected by a sign change using a Pattern window.

Consider the following diagram. The polyline is depicted as a bold black line. The dashed line around the polyline is the area within the distance of the radius. A, B, C, and D are incoming event positions.

Figure 4.1 *Polyline Relative to Several Event Positions*



The distance between event position A and the polyline is less than the radius. For that event, the Geofence window returns the polyline ID.

The event position B is close to the polyline when `strict-projection='false'`. In that case, the polyline ID is returned. B is out of the polyline proximity when `strict-projection='true'`. The polyline ID is not returned.

The event position C is out of the polyline proximity. The polyline ID is not returned.

With regard to event position D:

- P is the projected point of position D. The coordinates of P are returned by the fields specified by the `projection-fieldname` parameter.
- The distance between D and P is returned by the field specified by the `geodistance-fieldname` parameter.
- P to P3 to P2 to P1 is the distance to origin. That distance is returned by the field specified by the `distance-to-origin-fieldname` parameter.

- The segment number is 3, which is returned by the field specified by the `segmentnumber-fieldname` parameter.

For more information about the `projection-fieldname`, `geodistance-fieldname`, `distance-to-origin-fieldname`, and `segmentnumber-fieldname` parameters, see "output" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Circles

A *circle* encompasses the position of a location of interest. Define a circle with three values:

- two coordinates, X and Y (longitude and latitude), that represent the center of the circle
- a radius distance around the center

The following three fields of the input circle geometry window schema are required:

- A single key field of type Int64 or String. This field defines the ID of the circle.
- One of the following:
  - Two coordinate fields of type Double that contain the X and Y coordinates of the circle center
  - A data field of type Array(dbl), string, or rstring with coordinates defined as numbers in the X, Y order, separated by spaces

The schema can also contain the following optional fields:

- A radius field (double) that represents a circular area around the center point position. If you do not specify this field, the default distance defined by the parameter radius is used.
- A description field.

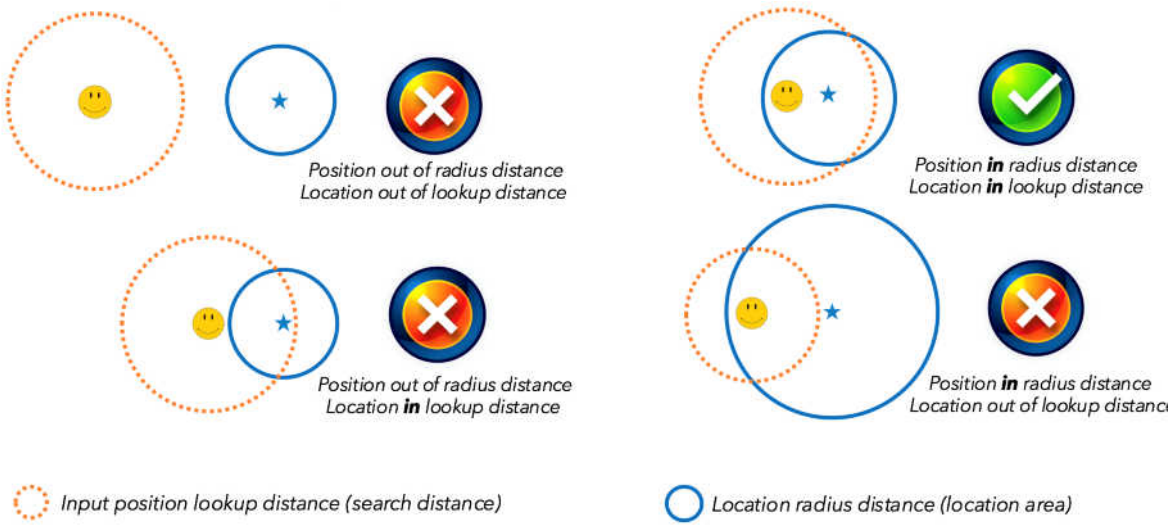
All other fields are ignored.

When working with circles, the Geofence window analyzes each event position that comes from the streaming window and returns the circle ID that matches the following criteria:

- If the position lookup distance is set to 0, then the position behaves like a simple point. It is either in or out of the circle. If it is in the circle, there is a match.
- Similarly, for circle geometries, if the circle radius is set to 0, then the circle behaves like a bare point. This point must be in the position lookup distance area to return a match.
- For any other values of the position lookup distance and the circle radius, the position and the circle's center must be within each other's distance. Otherwise, no match is returned. The position is within the circle and the distance between the circle's center and the position is lower than the lookup distance.
- If both the position lookup distance and the circle radius are equal to 0, then they must be the exact same point to have a match.

The position lookup distance is either defined by an additional event input field value or by the parameter `lookupdistance`.

Figure 4.2 *Circles Geometry Lookup Logic*



If there are multiple matching geometries in the lookup distance and if the option `output-multiple-results` is set to true, then multiple events are produced (one per geometry).

Positions Window Schema

The positions input window's schema can contain any type and number of fields that are propagated to the output schema. The following schema fields are used by the Geofence window:

- Two mandatory coordinate fields of type Double that contain the X and Y or longitude and latitude coordinates.
- An optional field of type Double that contains the lookup distance for the current position event. This field is used only by Geofence windows that are configured to use circles geometries.

All other fields are ignored and propagated to the output schema.

Output Schema

The Geofence window behaves like a lookup join, so its output schema is automatically defined by the following fields in the following order:

- All fields that come from the input position window in their respective order.

- A mandatory field of type int64 or string that receives the ID of the geometry. If no geometries are found, then the value of this field is null in the produced event. This field is defined by the parameter `geoid-fieldname`.

Output schema are appended with the following fields:

- A field that receives the description of the geometry when it exists in the geometry window schema. This field's presence and name are defined by the parameter `geodesc-fieldname`.
- A field (double) that receives the distance from the position to the geometry. The value of this field is one of the following:
  - the distance to the center of the circle
  - the distance to the closest segment of the polyline
  - the distance to the centroid or closest segment of the polygon

  This field's presence and name are defined by the parameter `geodistance-fieldname`.
- If `output-multiple-results` is set to true, then there is a mandatory key field of type Int64 that receives the event number of the matching geometry. This field's name is defined by the parameter `eventnumber-fieldname`.
- A field of type string that receives geometry type. This field's presence and name are defined by the parameter `geotype-fieldname`.
- Two fields (double) that receive the coordinates of the projected point on the closest segment of the polyline. These fields' values are null for geometry types other than polylines. These fields' presence and names are defined by the parameter `projection-fieldname`. These fields' names are prepended with _X and _Y respectively.
- A field (int32) that receives the segment number of the projected point on the closest segment of the polyline. Its value is null for geometry types other than polylines. This field's presence and name are defined by the parameter `segmentnumber-fieldname`.
- A field (array(double)) that receives the segments' length of the polyline. The sizes are in meters for geographic coordinates and in units for Cartesian coordinates. Its value is null for geometry types other than polylines. This field's presence and name are defined by the parameter `segmentsizes-fieldname`.
- A field (double) that receives the distance from the projected point to the first point of the polyline along the polyline. Its value is null for geometry types other than polylines. This field's presence and name are defined by the parameter `distance-to-origin-fieldname`.
- A list of fields that are propagated from the geometry input schema. These field's presence and names are defined by the parameter `include-geo-fields`.

Mesh Index

For fast and low latency lookup processing, the Geofence window implements an optimized mesh index algorithm that uses a spatial data structure. This structure subdivides space into buckets of grid shapes called *cells*. This structure is independent of the coordinate system in use, enabling the seamless use of any type of Cartesian, geographic, or projection coordinates space.

The mesh algorithm uses a parameter (called a *mesh factor*) that defines the scale of the space subdivision. This mesh factor is an integer within the [-5, 5] range that represents a power of 10 of the coordinate units in use.

For example, the default factor of 0 generates 1 subdivision per coordinate unit. A factor of 1 generates a subdivision per 10 units. A factor of -1 generates 10 subdivisions per unit. This factor can be set for both X and Y axes or independently for each axis.

Consider the following set of coordinates that represents a square polygon. Note the repeated first point at the end, closing the polygon:

```
[(1001.12,9500.12) (1001.12,9510.12) (1010.12,9510.12)
(1010.2,9500.12) (1001.12,9500.12)]
```

- With a mesh factor of 1, the Geofence window divides the coordinates by 10^1. This results in `[(100,950) (100,951) (101,950) (101,951)]` and creates four mesh cells for this geometry: (101-100+1)*(951-950+1) = 4.
- With a factor of 2, it creates (10-10+1)*(95-95+1) = 1 mesh cell.
- When the mesh factor is set to -1, the window creates 9,191 mesh cells. This results in an oversized mesh: (10101-10011+1)*(95101-95001+1)=91*101 = 9191.

For optimal performance, you must adapt the mesh factor to the spatial coverage and to the number of loaded geometries. When the number of mesh cells per geometry is too high, the ingestion of geometries is slowed and an oversized index is generated. When the number of mesh cells per geometries is too low, the lookup process is slowed, which affects stream performance and latency.

A dedicated parameter `max-meshcells-per-geometry` sets the maximum allowed mesh cells created per geometries to avoid creating an oversized mesh, which would generate useless intensive calculations. When a geometry exceeds this limit, it is rejected. When you intend to cover a very large area, consider setting a higher mesh factor or setting a higher maximum number of mesh cells per geometry.

The Geofence window provides an internal algorithm that automatically computes and sets an appropriate mesh factor by analyzing the first 1,000 geometries ingested. It is automatically active by default. To set the mesh factors manually, set the parameter autosize-mesh to false.

Example

```
<window-geofence name="geofence_win" index="pi_HASH">

  <geofence
     coordinate-type="geographic"
     log-invalid-geometry="true"
     output-multiple-results="true"
     output-sorted-results="false"
     max-meshcells-per-geometry="100"
     autosize-mesh="true"
  />

  <geometry
     data-fieldname="GEO_data"
     desc-fieldname="GEO_desc"
     x-fieldname="GEO_x"
     y-fieldname="GEO_y"
     radius-fieldname="GEO_radius"
     radius="0"
     data-separator=" "
  />

  <position
     x-fieldname="GPS_longitude"
     y-fieldname="GPS_latitude"
     lookupdistance="110"
  />

  <output
     geoid-fieldname="GEO_id_out"
     geodesc-fieldname="GEO_desc_out"
     eventnumber-fieldname="event_nb"
     geodistance-fieldname="GEO_dist"
  />

</window-geofence>
```

Using Procedural Windows

Overview

Use a Procedural window to specify an arbitrary number of input windows and input handler functions for each input window. You can define Procedural window

input handlers by using a callable C++ function in a shared library. Use the `cxx-plugin` XML element to define this type of input handler.

Note: The Procedural window does not support the use of SAS Micro Analytic Service modules and stores. That support is provided in the Calculate window.

When an input event arrives, the input handler registered for the matching window is called. Then the events produced by that input handler function are generated.

In order for the state of the Procedural window to be shared across handlers, an instance-specific context object (such as `dfESPpcontext`) is passed to the handler function. Each handler has full access to what is in the context object. The handler can store data in this context for use by other handlers, or by itself during future invocations.

For information about the XML elements associated with Procedural windows, see "window-procedural" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* and "XML Language Elements Relevant to Procedural Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models* .

Using C++ Window Handlers

Here is an example of the signature of a Procedural window handler written in C++.

```
typedef bool (*dfESPevent_func)(dfESPpcontext *pc,
              dfESPschema *is, dfESPeventPtr nep,
               dfESPeventPtr oep, dfESPschema *os,
              dfESPptrVect<dfESPeventPtr>&oe);
```

The procedural context is passed to the handler. The input schema, the new event, and the old event (in the case of an update) are passed to the handler when it is called. The final parameters are the schema of the output event (the structure of events that the Procedural window produces) and a reference to a vector of output events. It is this vector where the handler needs to push its computed events.

Only one input window is defined, so define only one handler function and call it when a record arrives.

```
// This handler functions simple counts inserts, updates,
//      and deletes.
// It generates events of the form "1,#inserts,#updates,
//      #deletes"
//
bool opcodeCount(dfESPpcontext *mc, dfESPschema *is,
                    dfESPeventPtr nep, dfESPeventPtr oep,
                    dfESPschema *os, dfESPptrVect
                    <dfESPeventPtr>& oe) {

    derivedContext *ctx = (derivedContext *)mc;
    // Update the counts in the past context.
    switch (nep->getOpcode()) {
        case dfESPeventcodes::eo_INSERT:
```

```
            ctx->numInserts++;
            break;
          case dfESPeventcodes::eo_UPDATEBLOCK:
            ctx->numUpdates++;
            break;
          case dfESPeventcodes::eo_DELETE:
            ctx->numDeletes++;
            break;
        }

      // Build a vector of datavars, one per item in our output
      //      schema, which looks like: "ID*:int32,insertCount:
      //      int32,updateCount:int32,deleteCount:int32"

  dfESPptrVect<dfESPdatavarPtr> vect;
  os->buildEventDatavarVect(vect);

  // Set the fields of the record that we are going to produce.

  vect[0]->setI32(1); // We have a key of only 1, we keep updating one
  record.
  vect[1]->setI32(ctx->numInserts);
  vect[2]->setI32(ctx->numUpdates);
  vect[3]->setI32(ctx->numDeletes);

  // Build the output Event, and push it to the list of output
  //      events.

  dfESPeventPtr ev = new dfESPevent();
  ev->buildEvent(os, vect, dfESPeventcodes::eo_UPSERT,
                 dfESPeventcodes::ef_NORMAL);
  oe.push_back(ev);

  // Free space used in constructing output record.
  vect.free();
  return true;
```

The following example shows how this fits together in a Procedural window:

```
<project name='project' pubsub='auto' threads='1'>
   <contqueries>
    <contquery name='contQuery' trace='proceduralWindow'>
     <windows>
      <window-source name='sourceWindow'>
       <schema>
        <fields>
         <field name='ID' type='int32' key='true'/>
         <field name='symbol' type='string'/>
         <field name='quantity' type='int32'/>
         <field name='price' type='double'/>
        </fields>
       </schema>
       <connectors>
        <connector class='fs' name='publisher'>
         <properties>
          <property name='type'>pub</property>
          <property name='fstype'>csv</property>
```

```
            <property name='fsname'>input.csv</property>
            <property name='transactional'>true</property>
            <property name='blocksize'>1</property>
           </properties>
          </connector>
         </connectors>
        </window-source>
        <window-procedural name='proceduralWindow'>
         <schema>
          <fields>
           <field name='ID' type='int32' key='true'/>
           <field name='insertCount' type='int32'/>
           <field name='updateCount' type='int32'/>
           <field name='deleteCount' type='int32'/>
          </fields>
         </schema>
         <cxx-plugin-context name='plugin'
function='get_derived_context'/>
         <cxx-plugin source='sourceWindow' name='plugin'
function='countOpcodes'/>
        </window-procedural>
       </windows>
       <edges>
        <edge source='sourceWindow' target='proceduralWindow'/>
       </edges>
      </contquery>
     </contqueries>
    </project>
```

Note: The `registerMethod` class consists of the following functions:

- `registerMethod_SO`
- `registerMethod_DSEXT`

For information about these functions, refer to the complete class and method documentation that is available at **`$DFESP_HOME/doc/html/index.html`**.

Whenever the Procedural window sees an event from the Source window (sw), it calls the handler opcodeCount with the context mc, and produces an output event.

An application can use the `dfESPengine::logBadEvent()` member function from a Procedural window to log events that it determines are invalid. For example, you can use the function to permit models to perform data quality checks and log events that do not pass. There are two common reasons to reject an event:

- The event contains a null value in one of the key fields.
- The opcode that is specified conflicts with the existing window index (for example, two Inserts of the same key, or a Delete of a non-existing key).

Converting DS2 Table Server Code to Run in SAS Micro Analytic Service Mode

Previous versions of SAS Event Stream Processing supported a Procedural window input handler that used DS2 code running in table server mode. Support for this functionality was deprecated at Release 5.2. To run DS2 code in a model, you now must define a SAS Micro Analytic Service module at the project level and a SAS Micro Analytic Service map in a Calculate window.

The key differences between running DS2 code in table server mode and running it in a SAS Micro Analytic Service map are as follows:

- The table server used a lazy binding on symbols. All input fields were always passed to the DS2 method. All declared variables in the DS2 method, provided that they matched a field name in the schema of the Procedural window, were exported to derived events.
- A SAS Micro Analytic Service module uses an interface similar to a function call, where only the input fields that you declare in the DS2 method signature are passed into the method. Only method parameters declared as `in_out` are exported from the DS2 method and then assigned to correspondingly named output fields. Any input field that matches an output field in name and type is echoed through the Calculate window. You do not need to explicitly pass matched fields to the DS2 method.
- A DS2 method running under the SAS Micro Analytic Service supports a read/write shared vector that can be shared across SAS Micro Analytic Service threads and packages. This enables fast concurrent data sharing when you use multiple threads.

The following example shows how to convert code running in table server mode to code that can run in a SAS Micro Analytic Service module. Suppose you have the following Source window schema:

```
id*:int64,sensorName:string,sensorValue:double
```

Suppose you have the following output window schema:

```
id*:int64,sensorName:string,sensorValue:double,absValue:double,sqrtValu
e:double
```

Suppose you have this DS2 code in table server mode:

```
ds2_options cdump;
    data esp.out;
    dcl double absValue;
    dcl double sqrtValue;
    method run();
       set esp.in;
         absValue  = abs(sensorValue);
         sqrtValue = sqrt(sensorValue);
       end;
    enddata;
```

You would use the following DS2 code in a SAS Micro Analytic Service module:

```
ds2_options sas;
    package p1/overwrite=yes;
       method compute(double sensorValue, in_out double absValue,
in_out double_sqrtValue);
         absValue  = abs(sensorValue);
         sqrtValue = sqrt(sensorValue);
       end;
    endpackage;
```

When your DS2 code processes opcodes, you must convert opcode values from integers to strings before you use it in a SAS Micro Analytic Service module.

Opcode	Table server mode (integer)	SAS Micro Analytic Service module (string)
Insert	1	insert
Update	2	update
Delete	3	delete
Upsert	4	upsert
Safe Delete	5	safedelete

You must also convert event flags.

Flag	Table server mode (integer)	SAS Micro Analytic Service module (string)
Normal	1	N
Retention	4	R

For example, consider the following DS2 code running in table server mode:

```
<ds2-tableserver source="Aggregate1">
    <code>
    <![CDATA[ds2_options cdump;
            data esp.out;
              method run();
                set esp.in;
                if _opcode = 2 or _opcode = 3 then
                   _opcode = 1;
              end;
            enddata;]]>
    </code>
</ds2-tableserver>
```

The following code performs the same evaluation and produces the same results:

```
<mas-module module="opcode_module" language="ds2" func-
names="convert_opcode">
  <code>
  <![CDATA[ds2_options sas;
  package opcode_module/overwrite=yes;
    method convert_opcode(varchar(16) _inOpcode, in_out varchar
_outOpcode);
       if (_inOpcode = 'delete' or _inOpcode = 'update') then
           _outOpcode = 'insert';
       else
           _outOpcode = _inOpcode;
      end;
  endpackage;]]>
  </code>
</mas-module>
```

Although it is function call based, DS2 code used in a SAS Micro Analytic Service module can produce zero, one, or more than one output event for each input event.

Using Object Tracker Windows

Overview of Object Tracker Windows

Use an Object Tracker window to perform multiple object tracking (MOT) in real time. *Multiple object tracking* is the process of accurately estimating the identity and position of multiple objects over time by using a model that is based on incoming observations. These observations are the output of an object detection model.

The challenges of multiple object tracking include scene clutter, target dynamics, intraclass and interclass variation, measurement noise, and frame rate. The Object Tracker window addresses these challenges by coupling trackers with detectors in a paradigm called tracking-by-detection. The Object Tracker window enables you to choose between two methods of tracking-by-detection: the intersection-over-union (IOU) method and the ByteTrack method.

Note: It is recommended that you use the ByteTrack method because in most cases it is more efficient than the IOU method.

For information about the XML elements that are associated with Object Tracker windows, see window-object-tracker and "XML Language Elements Relevant to Object-Tracker Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

The Computer Vision with ONNX example and Pose Estimation with ONNX example are complete projects that use an Object Tracker window. For more

information, see "Install Example Projects" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

Overview of the Intersection-over-Union Method of Tracking-by-Detection

The intersection-over-union (IOU) method tracks objects by using only detection data. Basically, IOU is a number that quantifies the degree of overlap between two bounding boxes. Using the IOU method enables incremental and event-oriented tracking. It has produced good results on the MOTChallenge, which provides a framework for multitarget tracking, and on the UA-DETRAC benchmarks, which consist of challenging video sequences that are captured from real-world traffic scenes. For more information about the IOU method, see Bochinski, Eiselein, and Sikora (2017).

The IOU method that is implemented by the Object Tracker window provides the following functionality:

- It uses a velocity vector, which represents the rate of change of an object's position, to predict the next position of a tracked object whenever it is missing in the next frame. The `velocity-vector-frames` parameter of the `tracker` element defines the number of frames to use to calculate the velocity vector.
- After the first tentative match with existing tracks and detection occurs (IOU > σIOU), any detections that remain unassociated are matched as follows: detections that have an IOU > σIOU2 with some unmatched tracks are matched with whatever unmatched track has the greater IOU.
- Remaining unmatched detections create a track only when their IOU with existing tracks is less than the value of the `iou-sigma-dup` parameter of the `tracker` element. This functionality avoids reassignments of double detections. Setting the `iou-sigma-dup` parameter to the default value of `0.0` disables this feature.
- Tracks begin with multiple lives. When a track is left unmatched with any detection on a frame, it loses a life. After several frames without matching a detection, the track dies. The `max-track-lives` parameter of the `tracker` element sets the life duration of tracks without detection.

Here is sample XML code for an Object Tracker window that highlights the previously referenced parameters:

```
<window-object-tracker pubsub="true" name="tracking">
     <tracker method="iou" score-sigma-low="0.3" score-sigma-
high="0.7" iou-sigma="0.6" iou-sigma2="0.3" iou-sigma-dup="0" velocity-
vector-frames="11" max-track-lives="60" min-track-length="0" track-
retention="0"/>
     <output mode="wide" tracks="65" velocity-vector="true"/>
     <input count="objCount" score="obj_%_score" label="obj_%_label"
coord-type="rect" x="obj_%_x" y="obj_%_y" width="obj_%_w" height="obj_
%_h"/>
</window-object-tracker>
```

Overview of the ByteTrack Method of Tracking-by-Detection

The Object Tracker window's ByteTrack method is an implementation of the ByteTrack algorithm, which is a popular tracking-by-detection algorithm. For more information about ByteTrack, see Zhang et al. (2022).

Both the Object Tracker window's IOU method and the Object Tracker window's ByteTrack method perform IOU calculations. The Object Tracker window's implementation of the ByteTrack algorithm uses IOU calculations for both the first association step and the second association step.

Here is sample XML code:

```
<window-object-tracker pubsub="true" name="tracking">
          <tracker method="bytetrack" track-thresh="0.5" high-
thresh="0.6" match-thresh="0.8" max-track-lives="60" velocity-vector-
frames="30" />
          <output mode="array" velocity-vector="true"  />
          <input group-by="cam_id"  coord-type="rect" count="objCount"
x="obj_x" y="obj_y" width="obj_w" height="obj_h" score="obj_score"
label="obj_label"/>
        </window-object-tracker>
```

Here is more information about some of the parameters in the example:

- A velocity vector represents the rate of change of an object's position. The vector predicts the next position of a tracked object whenever it is missing in the next frame. The `velocity-vector-frames` parameter of the `tracker` element defines the number of frames to use to calculate the velocity vector.
- The `track-thresh` parameter of the `tracker` element specifies the detection score threshold that defines whether a track is a low score track or a high score track. The value of `0.5` that is shown in this example is the default value for this parameter.
- The `high-thresh` parameter of the `tracker` element specifies the threshold above which new tracks are created for new detections. The value of `0.6` that is shown in this example is the default value for this parameter.
- The `match-thresh` parameter of the `tracker` element specifies a matching threshold for similarity assignments for high score tracks. The value of this parameter is an IOU distance, so a higher value is less restrictive. The value of `0.8` that is shown in this example is the default value for this parameter.
- Tracks begin with multiple lives. When a track is left unmatched with any detection on a frame, it loses a life. After several frames without matching a detection, the track dies. The `max-track-lives` parameter of the `tracker` element sets the life duration of tracks without detection.
- The `group-by` parameter of the `input` element specifies that a field called `cam_id` is used to group incoming events. The `group-by` parameter is optional and is useful when a single Object Tracker window is receiving events from

multiple cameras. The Object Tracker window instantiates a tracker for each camera ID.

Input Schema for Object Tracker Windows

The Object Tracker window uses detection data in order to perform multiple object tracking. Detection data is derived from applying an object detection model (for example, YOLO) to a stream of visual images.

The Source window that processes incoming detection data uses an input schema that consists of the following fields:

- The frame ID for the input image.
- (Optional). A blob representation of the image.
- The detected objects count for the event.
- For each detected object, the input schema also consists of the following fields:
  - The x coordinate of the bounding box center or the lower corner, respective to the origin of the coordinates. In object detection output, this coordinate often refers to the top left corner.
  - The y coordinate of the bounding box center or the lower corner, respective to the origin of the coordinates. In object detection output, this coordinate often refers to the top left corner.
  - The width of the bounding box or the x coordinate of the higher bounding box corner, respective to the origin of the coordinates. In object detection output, this coordinate often refers to the bottom right corner.
  - The height of the bounding box or the y coordinate of the higher bounding box corner, respective to the origin of the coordinates. In object detection output, this coordinate often refers to the bottom right corner.
  - (Optional). The detection score. The default value is 1.
  - The label of the detected object.
  - The object attributes.
  - When specifying keypoints, an array `n*k*(x)` where `n` is the number of detected objects, `k` is the number of keypoints in position, and `x` is the x coordinate.
  - When specifying keypoints, an array `n*k*(y)` where `n` is the number of detected objects, `k` is the number of keypoints in position, and `y` is the y coordinate.
  - When specifying keypoints, an array `n*k*(s)` where `n` is the number of detected objects, `k` is the number of keypoints in position, and `s` is the specified score.
  - When specifying keypoints, an array `n*k*(label_id)` where `n` is the number of detected objects, `k` is the number of keypoints in position, and `label_id` is the specified label.

- (Optional). A time stamp.
- All other fields are propagated to the output event, including the event key fields.

A *keypoint* identifies a specific, detectable feature within an image that is used to track an object as it moves across frames. Typically, keypoints are distinct points, corners, or edges that can be consistently identified across different points of view. After they are specified, keypoints are used to estimate the position and orientation of an object relative to its surroundings. For example, in a category-level object pose tracking system, keypoints could represent the vertices of a bounding box that surrounds an object.

The following input schema handles array input:

```
frame_id*:int64,image:blob,objCount:int32,obj_label:string,obj_score:ar
ray(dbl),
obj_x:array(dbl),obj_y:array(dbl),obj_w:array(dbl),obj_h:array(dbl),
obj_kpts_count:array(i32),obj_kpts_x:array(dbl),obj_kpts_y:array(dbl),o
bj_kpts_score:array(dbl),obj_kpts_label_id:array(i32),
time:stamp
```

The following Object Tracker window accepts input from a Source window that uses that input schema.

```
<window-object-tracker pubsub="true" name="tracker">
          <tracker method="iou" score-sigma-low="0.08" score-sigma-
high="0.08" iou-sigma="0.15" iou-sigma2="0.1" iou-sigma-dup="0.1"
velocity-vector-frames="5" max-track-lives="40" track-retention="10"/>
          <output mode="array" tracks="30" velocity-vector="true"
keypoints="true" scale-x="220" scale-y="640"/>
          <input count="objCount" score="obj_score" label="obj_label"
coord-type="rect" x="obj_x" y="obj_y" width="obj_w" height="obj_h"
                  kpts-count="obj_kpts_count" kpts-label-
id="obj_kpts_label_id" kpts-score="obj_kpts_score" kpts-x="obj_kpts_x"
kpts-y="obj_kpts_y"/>
</window-object-tracker>
```

The following input schema processes three detected objects:

```
frame_id*:int64,image:blob,objCount:int32,obj_0_x:double,obj_0_y:double
,obj_0_h:double,
obj_0_w:double,obj_0_score:double,obj_0_label:string,obj_1_x:double,obj
_1_y:double,obj_1_h:double,
obj_1_w:double,obj_1_score:double,obj_1_label:string,obj_2_x:double,obj
_2_y:double,obj_2_h:double,
obj_2_w:double,obj_2_score:double,obj_2_label:string,obj_3_x:double,obj
_3_y:double,obj_3_h:double,
obj_3_w:double,obj_3_score:double,obj_3_label:string,time:stamp
```

Use the input element of the Object Tracker window to define the naming conventions of the object's input fields. When processing non-array input, use the `%` character as a placeholder for the object number.

For example, if the `'x'` attribute of the `<input>` element has the value **`'obj_%_x'`**, then the window assumes that the x coordinate of the object is `obj_0_x, obj_1_x, obj_2_x, ....`

```
<window-object-tracker pubsub="true" name="tracking">
```

```
     <tracker method="iou" score-sigma-low="0.3" score-sigma-
high="0.7" iou-sigma="0.6" iou-sigma2="0.3" iou-sigma-dup="0" velocity-
vector-frames="11" max-track-lives="60" min-track-length="0" track-
retention="0"/>
     <output mode="wide" tracks="65" velocity-vector="true"/>
     <input count="objCount" score="obj_%_score" label="obj_%_label"
coord-type="rect" x="obj_%_x" y="obj_%_y" width="obj_%_w" height="obj_
%_h"/>
</window-object-tracker>
```

`coord-type=rect` specifies a bounding box by using the x and y coordinates of its top left corner along with width and height values. `coord-type=yolo` specifies a bounding box by using the x and y coordinates of its center along with width and height values. `coord-type=coco` specifies a bounding box by using the x-min and y-min coordinates of its top left corner along with x-max and y-max coordinates of its bottom right corner. This format takes its name from a data set known as COCO (common objects in context).

If the `<input>` element is not used, then the standard naming conventions for output fields of analytic stores are used.

Output Schema for Object Tracker Windows

Overview

The output of an Object Tracker window conforms to a wide schema, a long schema, or an array schema, depending on the value that you set for the `mode` attribute of the output element.

All output schemas include all non-object fields of the input window. The key fields of the input window are not part of the output schema.

Wide Schema

The wide schema includes the following fields:

*prefix*_density	type="int32" [1]
*prefixN*_id	type="int64" [2] [3]
*prefixN*_label	type="string"
*prefixN*_score	type="double"
*prefixN*_x	type="double"
*prefixN*_y	type="double"
*prefixN*_w	type="double"
*prefixN*_h	type="double"
*prefixN*_center_x	type="double" [4]
*prefixN*_center_y	type="double"
*prefixN*_track_x	type="array(dbl)" [5]
*prefixN*_track_y	type="array(dbl)"

```
prefixN_track_count          type="int32"
prefixN_track_attributes     type="string"
```

1 Use the `prefix` attribute of the `<output>` element to define the value of *prefix*. The default value of `prefix` is **`'Object'`**.

2 The value of *N* is the zero-based index of the object number. Use the `tracks` attribute of the `<output>` element to define the number of output objects.

3 The `<..>_id` field contains the identifier of the object's track.

4 The `<..>_center_x` and `<..>_center_y` fields contain the x and y coordinates of the center of the bounding box of the object's current position.

5 The `<..>_track_x` and `<..>_track_y` fields are array fields that contain the history of the x and y coordinates of the center of the bounding box, frame by frame. This information provides the full track of the object. That is, a line that links these points would trace the object's movement. For more information, see "Array Schema".

When `velocity-vector="true"`, the following additional fields appear in the schema:

```
prefixN_vvect_x     type="double" 1
prefixN_vvect_y     type="double"
```

1 The `<..>_vvect_x` and `<..>_vvect_y` fields contain the coordinates of the last velocity vector of the object.

For example, if `prefix='Object'`, `tracks=4`, and `velocity-vector="true"` and the input schema is the one that is specified in "Input Schema for Object Tracker Windows" , then the output schema looks like this:

```
frame_id*:int64,image:blob,objCount:int32,time:stamp,Object_density:int32,Object0_id:int64,Object0_label:string,
Object0_score:double,Object0_x:double,Object0_y:double,Object0_w:double,Object0_h:double,
Object0_center_x:double,Object0_center_y:double,Object0_track_x:array(dbl),Object0_track_y:array(dbl),
Object0_vvect_x:double,Object0_vvect_y:double,Object1_id:int64,Object1_label:string,Object1_score:double,
Object1_x:double,Object1_y:double,Object1_w:double,Object1_h:double,Object1_center_x:double,
Object1_center_y:double,Object1_track_x:array(dbl),Object1_track_y:array(dbl),Object1_vvect_x:double,
Object1_vvect_y:double,Object2_id:int64,Object2_label:string,Object2_score:double,Object2_x:double,
Object2_y:double,Object2_w:double,Object2_h:double,Object2_center_x:double,Object2_center_y:double,
Object2_track_x:array(dbl),Object2_track_y:array(dbl),Object2_vvect_x:double,Object2_vvect_y:double,
Object3_id:int64,Object3_label:string,Object3_score:double,Object3_x:double,Object3_y:double,
Object3_w:double,Object3_h:double,Object3_center_x:double,Object3_center_y:double,
Object3_track_x:array(dbl),Object3_track_y:array(dbl),Object3_vvect_x:double,Object3_vvect_y:double
```

When `keypoints="true"`, the following additional fields appear in the schema:

```
prefixN_track_kpts_count     type="array(i32)" 1
prefixN_track_kpts_x     type="array(dbl)" 2
prefixN_track_kpts_y     type="array(dbl)" 3
prefixN_track_kpts_score     type="array(dbl)" 4
prefixN_track_kpts_label_id     type="array(i32)" 5
```

1 An array `n*t*(k)` that specifies the number of keypoints for each track position.

2 An array `n*t*k*(x)` that specifies keypoints for each track position for the x coordinate.

3 An array `n*t*k*(y)` that specifies keypoints for each track position for the x coordinate.

4 An array `n*t*k*(s)` that specifies keypoint scores for each track position.

5 An array `n*t*k*(label_id)` that specifies keypoint label IDs for each track position.

Long Schema

The long schema includes the following fields:

```
prefix_density 1          type="int32"
prefix_id                 type="int64"
prefix_label              type="string"
prefix_score              type="double"
prefix_x                  type="double"
prefix_y                  type="double"
prefix_w                  type="double"
prefix_h                  type="double"
prefix_center_x           type="double" 2
prefix_center_y           type="double"
prefix_track_x            type="array(dbl)" 3
prefix_track_y            type="array(dbl)"
prefix_track_count             type="int32"
prefix_track_attributes        type="string"
```

1 Use the `prefix` attribute of the `<output>` element to define the value of *prefix*. The default value of `prefix` is **`'Object'`**.

2 The `<..>_center_x` and `<..>_center_y` fields contain the x and y coordinates of the center of the bounding box of the object's current position.

3 The `<..>_track_x` and `<..>_track_y` fields are array fields that contain the history of the x and y coordinates of the center of the bounding box, frame by frame. This information provides the full track of the object. That is, a line that links these points would trace the object's movement. For more information, see "Array Schema".

When `velocity-vector="true"`, the schema contains the following additional fields:

```
prefix_vvect_x         type="double"
prefix_vvect_y         type="double"
```

For example, suppose that `prefix='Object'`, `velocity-vector="true"`, and the input schema is the one that is specified in "Input Schema for Object Tracker Windows" . The output schema looks like this:

```
frame_id*:int64,image:blob,objCount:int32,time:stamp,Object_density:int
32,Object_id*:int64,Object_label:string,
Object_score:double,Object_x:double,Object_y:double,Object_w:double,Obj
ect_h:double,
Object_center_x:double,Object_center_y:double,Object_track_x:array(dbl)
,Object_track_y:array(dbl),
Object_vvect_x:double,Object_vvect_y:double
```

When `keypoints="true"`, the following additional fields appear in the output schema:

```
prefix_track_kpts_count    type="array(i32)" 1
prefix_track_kpts_x    type="array(dbl)" 2
prefix_track_kpts_y    type="array(dbl)" 3
prefix_track_kpts_score    type="array(dbl)" 4
prefix_track_kpts_label_id    type="array(dbl)" 5
```

1 An array `t*(k)` that specifies the number of keypoints for each track position.

2 An array `t*k*(x)` that specifies keypoints for each track position for the x coordinate.

3 An array `t*k*(y)` that specifies keypoints for each track position for the x coordinate.

4 An array `t*k*(s)` that specifies keypoints scores for each track position.

5 An array `t*k*(label_id)` that specifies keypoints label IDs for each track position.

Array Schema

Assume that `n` equals the number of objects tracked and `t` equals the number of positions in the track. The array schema includes the following fields:

```
prefix_density    type="int32" 1
prefix_id          type="array(i64)"  2
prefix_label       type="string" 3
prefix_score       type="array(dbl)" 4
prefix_x           type="array(dbl)" 5
prefix_y           type="array(dbl)" 6
prefix_w           type="array(dbl)" 7
prefix_h           type="array(dbl)" 8
prefix_center_x    type="array(dbl)" 9
prefix_center_y    type="array(dbl)" 10
prefix_track_x     type="array(dbl)" 11
prefix_track_y     type="array(dbl)" 12
prefix_track_count type="array(i32)" 13
prefix_attributes  type="string" 14
```

1 `n`, which specifies the number of objects.

2 An array of `n*(id)`.

3 `n*(label)` is a CSV string of labels, one per object.

4 An array of `n*(score)`.

5 An array of `n*(x)`.

6 An array of `n*(y)`.

7 An array of `n*(w)`.

8 An array of `n*(h)`.

9 An array of `n*(c_x)`.

10 An array of `n*(c_y)`.

11 An array of `n*t*(x)`.

12 An array of `n*t*(y)`.

13 `n*(t)`, which is the number of positions in track for each object.

14 `n*(attrList)` is a CSV as a string or rstring with `','` as the separator (label-separator attribute).

When `velocity-vector="true"`, the output schema contains the following additional fields:

```
prefix_vvect_x        type="double"
prefix_vvect_y        type="double
```

When `keypoints="true"`, the following additional fields appear in the output schema:

```
prefix_track_kpts_count    type="array(i32)" 1
prefix_track_kpts_x      type="array(dbl)" 2
prefix_track_kpts_y      type="array(dbl)" 3
prefix_track_kpts_score      type="array(dbl)" 4
prefix_track_kpts_label_id      type="array(i32)" 5
```

1 An array `t*(k)` that specifies the number of keypoints for each track position.

2 An array `n*t*k*(x)` that specifies keypoints for each track position for the x coordinate.

3 An array `n*t*k*(y)` that specifies keypoints for each track position for the x coordinate.

4 An array `t*k*(s)` that specifies keypoints scores for each track position.

5 An array `t*k*(label_id)` that contains keypoint label IDs for each track position.

The Pose Estimation with ONNX example uses an array schema in the w_object_tracker_array window. This window's definition includes `keypoints="true"` and `velocity-vector="true"`. Here is the array schema output for that window:

```
id:int64,_nObjects_:int64,image:blob,Object_density:int64,Object_id:arr
ay(i64),Object_label(string),
Object_score:array(i64),Object_x:array(i64),Object_y:array(i64),Object_
w:array(i64),Object_h:array(i64),
Object_center_x:array(i64),Object_center_y:array(i64),Object_track_x:ar
ray(i64),Object_track_y:array(i64),
Object_track_count:array(i64),Object_attibutes:string,Object_vvect_x:ar
ray(i64),Object_vvect_y:array(i64),
Object_track_kpts_count:array(i64),Object_track_kpts_x:array(i64),Objec
t_track_kpts_y:array(i64),
Object_track_kpts_score:array(i64),Object_track_kpts_label_id:array(i64
)
```

Here is a part of the output for an event in the Pose Estimation with ONNX example. One object's x coordinate and y coordinate have been tracked 10 times:

```
Object_track_x:
[1,057;1,056;1,056;1,056;1,056;1,056;1,056;1,057;1,057;1,057]
Object_track_y: [329;329;329;329;330;330;330;330;331;331]
Object_track_count: [10]
```

Here is another event in the Pose Estimation with ONNX example. There are two objects. The event includes 10 positions for the first object and 10 positions for the second object.

```
Object_track_x:
[1,057;1,057;1,058;1,058;1,058;1,057;1,057;1,057;1,057;1,055;39;55;68;6
9;80;84;85;87;86;86]
Object_track_y:
[331;332;332;332;332;333;333;333;332;332;399;399;402;399;389;394;379;36
0;359;350]
Object_track_count: [10;10]
```

In `Object_track_x`, the first 10 positions `(1,057;1,057;1,058;1,058;1,058;1,057;1,057;1,057;1,057;1,055)` relate to the first object. The positions are ordered from the oldest to the most recent: `1,057` (the first number) is the oldest position and `1,055` is the most recent. The other 10 positions relate to the second object.

Managing the History of the Object Tracker Window

Use the `track-retention` parameter to specify the number of frames that tracks keep in the history of positions in memory. By default, the value of this parameter is `0`. When the same object persists in an image over hours or days, leaving this parameter with a value of `0` potentially grows the track history indefinitely. This growth has the potential to crash a project.

You can specify under what conditions the history of the Object Tracker window should be cleared so that subsequent tracking can proceed in a fresh state.

Use the `frame-skip-field` parameter to specify the name of the field in the input event that contains a frame index or a time to use to clear the history. Use the `frame-skip-amount` parameter to specify the number of frames that can be missed or the time in seconds that can transpire before history is cleared and object tracking is restarted.

For example, assume that you have a stream of input events with the following fields and contents:

Table 4.6 *Fields and Contents of Sample Input Events*

frame_number	image
1	image1
2	image2
3	image3
10	image10

Assume that you set `"frame-skip-field"="frame_number"`. Further assume that the `frame_number` field contains a frame index and that `"frame-skip-amount"=5`.

Because the gap between `"frame_number"=3` and `"frame_number"=10` is greater than the specified frame skip amount of `5`, the objects that are tracked in the previous images (**`image1`**, **`image2`**, and **`image3`**) are cleared. Object tracking restarts with **`image10`**.

For information about the relevant XML syntax, see "XML Language Elements Relevant to Object-Tracker Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

References

Bockinski, Erik, Volker Eiselein, and Thomas Sikora. 2017. "High-Speed Tracking-by-Detection Without Using Image Information." In *IEEE International Conference on Advanced Video and Signal-based Surveillance*. Lecce, Italy. IEEE Computer Society. http://elvera.nue.tu-berlin.de/files/1517Bochinski2017.pdf

Zhang, Yifu, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, Xinggang Wang. 2022. "ByteTrack: Multi-Object Tracking by Associating Every Detection Box." https://arxiv.org/pdf/2110.06864

Using StateDB Windows

Overview

Join operations, aggregations, and rolling aggregations can be performed with very high throughput and very low latency in SAS Event Stream Processing projects. Typically, the data that is required to accomplish those tasks is stored in RAM. This data represents the "state" of the window that performs those tasks.

Sometimes the volume of this "state" data can tax system capabilities. For example, when joins are performed over a huge volume of data or when an aggregation period spans days or months, processing performance can suffer. For example:

- The amount of RAM that is required can exceed system capabilities.
- The time to load data in memory can be unacceptably slow.
- Ordinarily, a large amount of data is reprocessed with failover. Reprocessing can lead to the system recovering from failure at an intolerably slow rate.

- Stateful models can lead to slower auto-scaling because each node must be loaded with the required state data. This slower auto-scaling can result in unnecessary duplication of data and an increased in-memory footprint.

The StateDB Writer and StateDB Reader windows enable you to use an externally deployed, high-performance data store or database to store and maintain the "state" that is required for joins and aggregations. Using an external store reduces the need to use RAM. Here are some of the benefits:

- These external stores can manage large volumes of data over time more efficiently than SAS Event Stream Processing can.
- Projects are as stateless as possible as a result of data being stored externally.
- Data can be shared across pods and projects, which improves scalability.
- There is no need to preload data whenever a project starts.
- For failover, there is no need to reload data because it is immediately available and is never lost.

IMPORTANT You must download and install Redis or install SingleStore in order to use StateDB windows. For more information, see the Redis website or the SingleStore website.

IMPORTANT When a StateDB Reader window reads from a Singlestore table, that table must contain the following two fields:

- `esp_meta_timestamp`
- `esp_meta_ttd`

Those fields are used by SAS Event Stream Processing to manage deleting fields from the Singlestore table. They are automatically created when the table is created by a StateDB Writer window. When you use a table that was otherwise created, you must manually add those two fields as type BIGINT.

For information about the XML elements that are associated with the StateDB windows, see "XML Language Elements Relevant to StateDB Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Using the StateDB Writer Window

Use the StateDB Writer window to complete these actions:

- write an incoming event to the defined external database
- create required indexes for the event
- pass the input event unchanged to the next window of a project

Within a pod, writing is synchronous in order to keep operations properly sequenced within a project. The key fields that are specified are used as the primary index within the database. All other fields that are selected from the input mapping are written using the specified name.

The StateDB Writer window stores data (events) as key-value pairs. In order to enable querying on non-key fields without degrading performance, the StateDB Writer enables you to create secondary indexes on those fields. These secondary indexes are mandatory in order to perform one-to-many joins or aggregations.

Using the StateDB Reader Window

Use the StateDB Reader window to query or fetch data from the defined external database based on the incoming event and specified criteria. These criteria are based in part on the indexes that you created in the StateDB Writer window.

You can fetch zero, one, or multiple records from the database. You can configure window properties to aggregate those records. The window then produces one or multiple events as output.

Note: The StateDB Reader window generates an error whenever the StateDB Reader query specifies fields that do not have a secondary index or a primary index that exists on those fields.

The StateDB Reader window supports the following functionality.

Table 4.7 *Supported Functionality*

Functionality	Description
Lookup operation	The functionality is similar to a one-to-one join that is typically performed by a Join window. For every incoming event in the StateDB Reader window, there is zero or one matching record in the database based on the primary index. When there are no matching records, the StateDB Reader window writes a null field. When there are matching records, the StateDB Reader window writes the value of the field itself.
Fetch multiple events	The functionality is similar to a one-to-many join that is typically performed by a Join window. You must specify an additional key, which comes from the database (`source=query`) and must be unique for the fetched group of events. It is recommended to add one or more extra keys from the fields that are fetched from the database in order to create unique keys for each outgoing event per single incoming event.

Functionality	Description
	For every incoming event in the StateDB Reader window, there are zero or more matching records in the database based on a secondary index. When there are no matching records, the StateDB Reader window writes a blank field. When there are matching records, the StateDB Reader window writes a transaction block that contains multiple events with fields from the events that are fetched from the database.
Aggregate events	The functionality is similar to the aggregations that are performed by the Aggregate window. For every incoming event there is a single output event. The output event contains fields that are defined by the aggregate functions. You can use the following aggregate functions in the StateDB Reader window: ■ `ESP_aAve(value)` ■ `ESP_aCount()` ■ `ESP_aFirst(value)` ■ `ESP_aLastNonNull(value)` ■ `ESP_aMax(value)` ■ `ESP_aMin(value)` ■ `ESP_aLag(value, lag)` ■ `ESP_aLast(value)` ■ `ESP_aCat(value, stringConstant)` ■ `ESP_aSum(value)` Configure these functions in the output schema of the StateDB Reader window in SAS Event Stream Processing Studio. For more information about aggregate functions, see “Aggregate Functions for Aggregate Window Field Calculation Expressions” .

Using Transport Layer Security (TLS) in StateDB Windows When Redis Is the External Data Store

You can use Transport Layer Security (TLS) to secure the connection between a Redis data store and SAS Event Stream Processing. By default, the connection is not secure.

SAS Event Stream Processing ships with OpenSSL software, which it uses to authenticate a Redis server. OpenSSL software uses a specific logic to search for

certificates and other relevant artifacts. SAS Event Stream Processing uses these certificates and artifacts for authentication purposes.

Client-side keys and certificates are necessary only when mutual authentication is required.

To secure the connection:

1 Specify `use-ssl="true"` in the `statedb` element of the project XML code. Alternatively, set the parameter in SAS Event Stream Processing Studio.

2 If necessary, set values for the following properties of the StateDB window:

Attribute	Description
[`ca-cert-filename=`*`path`*]	Specifies the name of a CA certificate or a bundle file to load and use for validation. A bundle file specifies a set of known CA certificates.
[`ca-path=`*`path`*]	Specifies the directory where trusted CA certificate files are stored in an OpenSSL-compatible structure. **Note:** If you specify a path, then TLS uses whatever authentication artifacts are located there. You need not specify the name of a CA certificate or bundle file.
[`cert-filename=`*`name`*]	Specifies the name of a client-side certificate. When specified, `private-key-filename` must also be specified.
[`private-key-filename=`*`name`*]	Specifies the name of private key files to use for authentication. When specified, `cert-filename` must also be specified.
[`server-name=`*`name`*]	Server Name Indication (SNI) is an extension to the Transport Layer Security (TLS) protocol that enables a client to indicate which host name it is attempting to connect to. This attribute specifies the SNI TLS extension.

Using Secondary Indexes in StateDB Windows When Redis Is the External Data Store

Because Redis does not provide out-of-box support for secondary indexes, you must use the StateDB Writer window to implement one. To understand how this secondary index works, you must understand how Redis implements primary indexes.

Consider the following event schema:

```
assetid*:string, tagid*:string, timestamp*:stamp, value:double
```

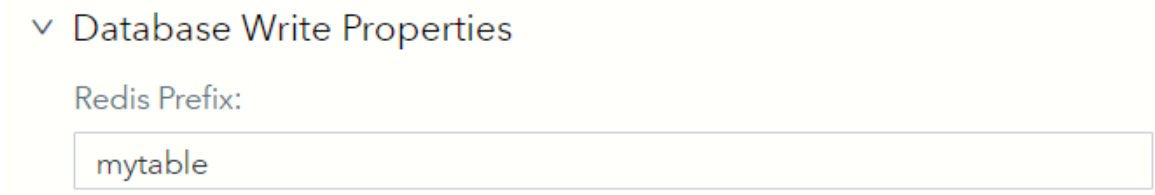
Note that `assetid`, `tagid`, and `timestamp` are key fields.

To get started with secondary indexes, specify the use of Redis as the external data store in the StateDB Writer window. Here, a single data store is used.

Figure 4.3 *Specify the Database Connection as Redis*

Suppose that you set the value of the Redis table prefix to `mytable`:

Figure 4.4 *Specify the Redis Prefix*



Now consider the following two events:

```
asset2, tagId46122, 1617765990190749, 0.461220
asset2, tagId186548, 1617859302543312, 0.461333
```

Redis stores events as key-value records in a Redis hash structure. The primary index keys are stored as follows:

`prefix_name:key_value[,key_value,...]`.

Therefore, when the StateDB Writer window processes the specified events, they have primary index keys as follows:

```
mytable:asset2:tagId46122:1617765990190749
mytable:asset2:tagId168548:1617859302543312
```

Redis stores secondary indexes as key-value records in a Redis Sorted Set structure. The keys of this secondary index are stored as follows: *`prefix_name`*`:SK:`*`sec_key_value`*`[,`*`sec_key_value`*`,...]`. The values are the related primary indexes of this group.

In the StateDB writer window, suppose that you define a secondary index on the `mytable` prefix as follows:

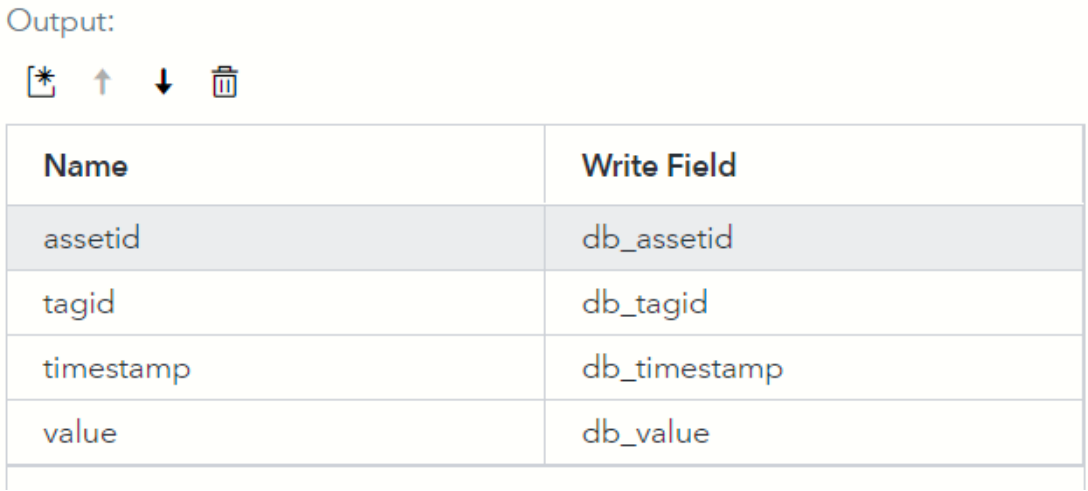
Figure 4.5 *Specify a Secondary Index*

Redis stores this secondary index key as follows:

```
mytable:SK:sec_idx:asset2
```

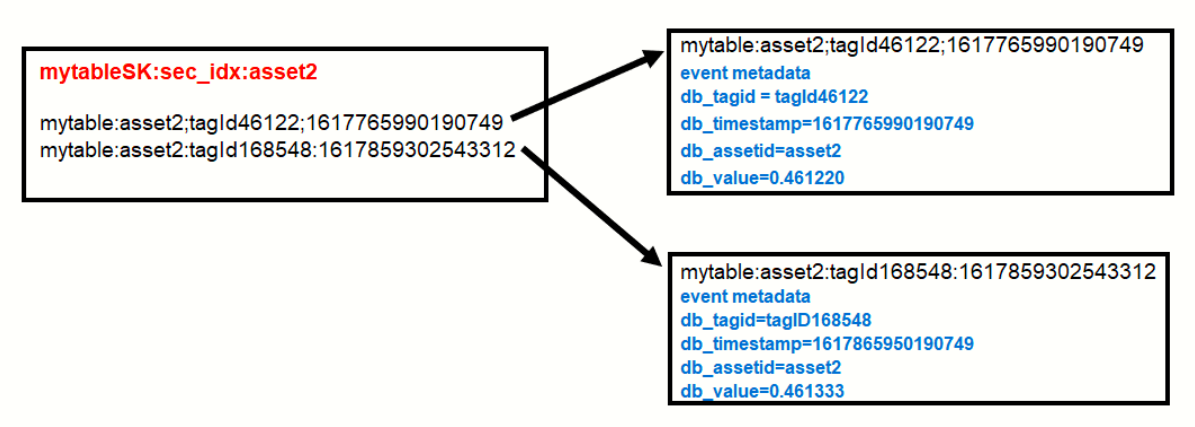
Suppose that you specify the output for the StateDB Writer window as follows:

Figure 4.6 *Specify Output of StateDB Writer Window*



Subsequently, each Redis entry points to a key value store which in turn stores the data for corresponding events.

Figure 4.7 *Correspondence of Indexes to Events*



Suppose that you have set up the StateDB Writer window to pass events to the StateDB Reader window.

Specify that the query performed by the StateDB Reader window uses the same table prefix that you defined for the StateDB Writer window (`mytable`). Next, define the actual query as follows:

- Specify `assetid` in the **Input Field**. This value matches the value that you specified for the secondary index `sec_idx`.
- Specify `db_assetid` in the **Redis Field**, which matches what you specified as the **Write Field** in the StateDB Writer window.
- When database type is Redis, the only available operator is `==`.

Figure 4.8 *Specify the Database Query*

Using this query, the StateDB Reader window fetches the list of primary indexes for the secondary index. Next, the StateDB Reader window fetches all events that match the secondary index `assetid` in the input event.

Because the secondary index field value of an incoming event is `asset2`, the following actions occur:

1 The StateDB Reader window fetches the Sorted Set with key as **`test:SK:sec_Idx:asset2`**, which gives the window a list of two keys:

```
mytable:asset2:tagId46122:1617765990190749

mytable:asset2: tagId186548:1617859302543312
```

2 The StateDB Reader window fetches the corresponding events that are stored in Hash data structure for the preceding two keys from Redis and return the following values:

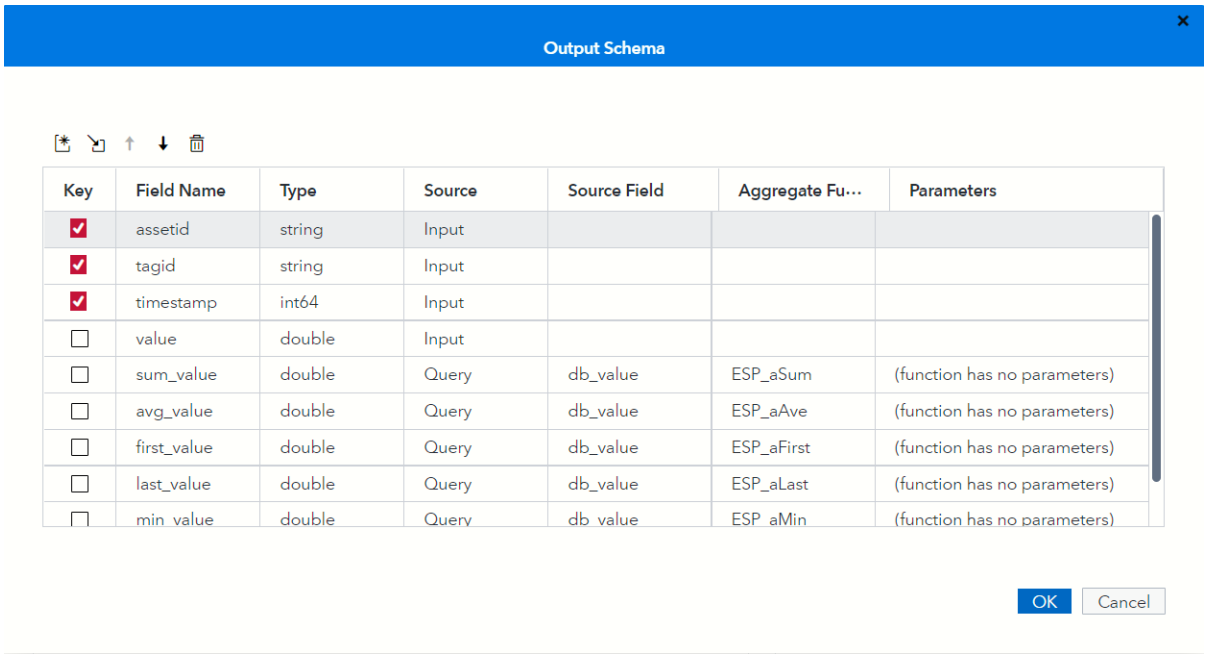
```
db_assetid=asset2, db_tagid=tagId46122,
db_timestamp=1617765990190749, db_value=0.461220

db_assetid=asset2, db_tagid=tagId186548,
db_timestamp=1617859302543312, db_value=0.461333
```

In the StateDB Reader window, create an output schema as follows:

- Specify **Source** to be **`Input`** for the first four specified fields. The fields come directly from the input event that was processed by the Source window.
- Specify **Source** to be **`Query`** for the remaining fields. Input values are matched with values that are fetched from the Redis data store.
- Select aggregation functions in the **Aggregation Function** to apply to the fetched value of **Source Field**. The results are written to the specified **Field Name**.

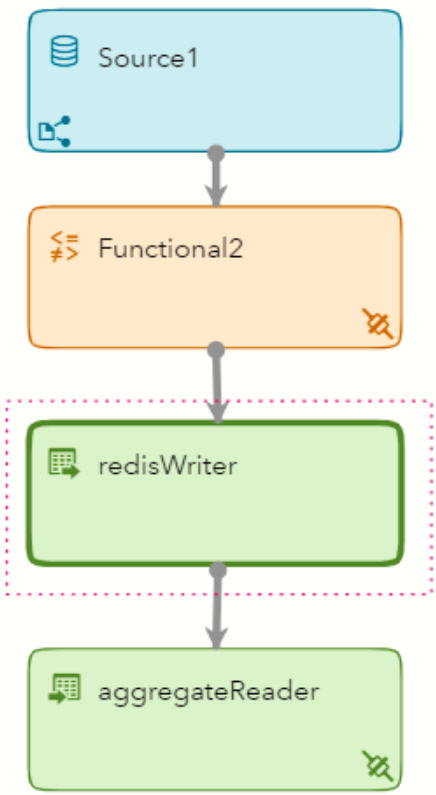
Figure 4.9 *Create an Output Schema*



Example: Flow of Events through the StateDB Writer and StateDB Reader Windows

Consider the following project, which implements a simple scenario for how events can flow through a project with one StateDB Writer window and one StateDB Reader window.

Figure 4.10 *Sample Project*



Suppose that the Source window receives data from industrial assets and contains the following input schema:

```
<schema>
     <fields>
        <field name="id" type="int64" key="true"/>
        <field name="assetid" type="string"/>
        <field name="tagid" type="string"/>
        <field name="timestamp" type="int64"/>
        <field name="value" type="double"/>
     </fields>
</schema>
```

As the figure shows, the Functional window receives events from the Source window and generates two new variables: time-to-live (`ttl`) and timestamp (`currentTimestamp`).

```
<window-functional index="pi_EMPTY" name="Functional2" pubsub="true">
          <schema>
            <fields>
              <field name="assetid" type="string" key="true"/>
              <field name="tagid" type="string" key="true"/>
              <field name="timestamp" type="int64" key="true"/>
              <field name="value" type="double"/>
              <field name="ttl" type="int64"/>
              <field name="currentTimestamp" type="int64"/>
            </fields>
          </schema>
          <function-context>
            <functions>
              <function name="currentTimestamp"><!
[CDATA[systemMicro()]]></function><!-- 1 -->
              <function name="ttl"><![CDATA[86400]]></function>
            </functions>
          </function-context>
     <connectors>
            <!-- 2 -->...
     </connectors>
</window-functional>
```

1 For more information about the `systemMicro` function, see Chapter 21, "Functional Window and Notification Window Functions" .

2 Use a File and Socket connector to verify output by writing it to a CSV file. (The actual code is not shown.)

The StateDB Writer window receives events from the Functional window.

Here is a StateDB Writer window using Redis.

```
<window-statedb-writer name="redisWriter" pubsub="true" >
  <statedb type="redis" hostname="generic.localhost" port="6379"
cluster="false" username="username" password="password" max-reconnect-
retries="20" reconnect-retry-delay="1000"/> <!-- 1 -->
    <write prefix="mytable" ttl="ttl" ttl-offset="60" del-dead-sec-
keys="true"> <!-- 2 -->
      <secondary-indexes>
        <secondary-index name="sec_idx"> <!-- 3 -->
          <fields>
            <field name="assetid"/>
          </fields>
        </secondary-index>
      </secondary-indexes>
      <output> <!-- 4 -->
        <field-selection name="assetid"    db-name="db_assetid" />
        <field-selection name="tagid"      db-name="db_tagid" />
        <field-selection name="timestamp"  db-name="db_timestamp" />
        <field-selection name="value"      db-name="db_value" />
      </output>
    </write>
</window-statedb-writer>
```

1 The window accesses a Redis database on `generic.localhost` on port 6379 with the user name **`username`** and the password **`password`**. Gather similar information about your specific database before you create your project.

2 The `prefix` value **`mytable`** is used to group data. Redis does not have table objects, so this prefix serves as the table name.

3 A secondary index `sec_idx` is created on the Redis database. When the StateDB Write window creates a secondary index, the index builds a structure of key-value pairs. The values of the secondary index field serve as keys, and a list of the primary keys of actual events serve as values. Here, there is a single secondary key called `assetid`, but you can create multiple secondary keys as needed.

4 The selected event's fields are written to the database with names that are specified by the `db-name` attribute values. For example, `assetid` is written as `db_assetid`. If the `db-name` attribute is not specified, then the StateDB Writer window uses the selected field name.

Note: Redis does not have native secondary index support. You must use the StateDB Writer window to create secondary index structures.

Here is a StateDB Writer window that uses SingleStore:

```
<window-statedb-writer name="SSWriter" pubsub="true" >
  <statedb type="singlestore" hostname="generic.localhost" port="3306"
username="username" password="password" database="memsql_example"/>
<!--1-->
    <write table="mytable" ttl="ttl" ttl-offset="60" del-dead-sec-
keys="true"> <!--2-->
      <secondary-indexes>
        <secondary-index name="sec_idx">
          <fields>
            <field name="assetid"/>
          </fields>
        </secondary-index>
      </secondary-indexes>
      <output>
        <field-selection name="assetid"    db-name="db_assetid" />
        <field-selection name="tagid"      db-name="db_tagid" />
        <field-selection name="timestamp"  db-name="db_timestamp" />
        <field-selection name="value"      db-name="db_value" />
      </output>
    </write>
</window-statedb-writer>
```

1 The window accesses a SingleStore database named "`memsql_example`" on `generic.localhost` on port 6379 with the user name `username` and the password `password`. Gather similar information about your specific database before you create your project.

2 Data is written to a table named `mytable`.

The StateDB Reader window aggregates incoming values (from the StateDB Writer window output) based on the secondary index `sec_idx`.

Here is a StateDB Reader window using Redis.

```
<window-statedb-reader name="aggregateReader" pubsub="true" >
  <statedb type="redis" hostname="generic.localhost" port="6379"
cluster="false" username="username" password="password" max-reconnect-
retries="20" reconnect-retry-delay="1000"/>
    <query prefix="mytable" query="assetid==db_assetid"/> <!--1 -->
      <schema>
        <fields>
          <field name="assetid" type="string" key="true"/>
          <field name="tagid" type="string" key="true"/>
          <field name="timestamp" type="int64" key="true"/>
          <field name="value" type="double"/>
          <field name="sum_value" type="double"/>
          <field name="avg_value" type="double"/>
          <field name="first_value" type="double"/>
          <field name="last_value" type="double"/>
          <field name="min_value" type="double"/>
          <field name="max_value" type="double"/>
          <field name="lag1_value" type="double"/>
        </fields>
      </schema>
      <output>
        <field-selection name="assetid" source="input"/> <!--2-->
        <field-selection name="tagid" source="input"/>
        <field-selection name="timestamp" source="input"/>
        <field-selection name="value" source="input"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aSum"/> <!--3-->
        <field-selection name="db_value" source="query"
aggregate="ESP_aAve"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aFirst"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aLast"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aMin"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aMax"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aLag,1"/>
      </output>
      <connectors>
       ...<!--4-->
      </connectors>
</window-statedb-reader>
```

1 The query is specified in the `query` tag as `query="assetid==db_assetid"`. The window matches the value of "`assetid`" with the value that was fetched from the Redis "`db_assetid`."

2 This field and the following three `field-selection` tags specify `source=input`. The values of those four fields come directly from the input event that was processed by the Source window.

3 This field and the following six `field-selection` tags specify `source=query`. The values for these six fields are fetched from the database. In this case, all six fields fetch the database field **`db_value`** and aggregate fetched values by using the specified aggregate function.

4 Use a File and Socket connector to verify output by writing it to a CSV file. (The actual code is not shown.)

Here is a StateDB Reader window using SingleStore. It is virtually identical to its Redis counterpart.

```
<window-statedb-reader name="aggregateReader" pubsub="true" >
  <statedb type="singlestore" hostname="generic.localhost" port="3306"
username="username" password="password" database="memsql_example"/>
    <query table="mytable" query="assetid==db_assetid"/>
      <schema>
        <fields>
          <field name="assetid" type="string" key="true"/>
          <field name="tagid" type="string" key="true"/>
          <field name="timestamp" type="int64" key="true"/>
          <field name="value" type="double"/>
          <field name="sum_value" type="double"/>
          <field name="avg_value" type="double"/>
          <field name="first_value" type="double"/>
          <field name="last_value" type="double"/>
          <field name="min_value" type="double"/>
          <field name="max_value" type="double"/>
          <field name="lag1_value" type="double"/>
        </fields>
      </schema>
      <output>
        <field-selection name="assetid" source="input"/>
        <field-selection name="tagid" source="input"/>
        <field-selection name="timestamp" source="input"/>
        <field-selection name="value" source="input"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aSum"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aAve"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aFirst"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aLast"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aMin"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aMax"/>
        <field-selection name="db_value" source="query"
aggregate="ESP_aLag,1"/>
      </output>
      <connectors>
       ...
      </connectors>
</window-statedb-reader>
```

Here are other scenarios that use StateDB windows:

- A single project can contain one StateDB Writer window and multiple StateDB Reader windows.
- There can be one StateDB Writer window in one project that communicates with other projects that contain one or more StateDB Reader windows.

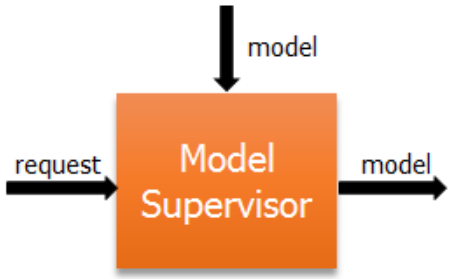
- You can use alternative methods to populate Redis with state data (for example, the Redis CLI). You can then create one or more projects with one or more secondary indexes exist in the database, you can query only on the Redis primary key. Only lookup operations are supported; one-to-many joins and aggregations are not supported.

5

Analytics

Using Model Supervisor Windows

Model Supervisor windows manage the flow of model events. Through input request events, you can control what model to deploy and when and where to deploy it. Model events are published to Score windows.



A Model Supervisor window can receive any number of model events. In a streaming analytics project, model events are typically sent by a Train window or a Model Reader window. After receiving a model event, a Model Supervisor window processes and publishes events to other streaming analytics windows based on the Model Supervisor window's deployment mode and on user requests.

Table 5.1 *Properties of window-model-supervisor element*

Property	Required or Optional	Description
`name`	Required	Specifies the window name. It must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
`deployment-policy`	Optional	Specifies the deployment policy of the offline models. Valid options are **`immediate`**, which sends model events to any receiving window immediately after receiving item; and **`on-demand`**, which sends model events according to the requests specified at the command line.
`capacity`	Optional	Specifies the maximum number of offline models to allow. After the capacity is reached, older model events are discarded.
`pubsub`	Optional	Specifies whether to publish and subscribe to the window. When the project-level value of `pubsub` is **`manual`**, a value of **`true`** enables publishing and subscribing and a value of **`false`** disables it.

The following is a generic example of a Model Supervisor window element:

```
<window-model-supervisor name='w_supervisor' deployment-policy='on-
demand' capacity='1000'>
     <connectors>
     ...
     </connectors>
</window-model-supervisor>
```

For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

Using Model Reader Windows

In most cases, Model Reader windows receive request events that reference the location and type of an offline model. Offline models are specified, developed, trained, and stored separately from the ESP server.

Then, Model Reader windows publish model events that contain model metadata to Score windows or to Model Supervisor windows.

For recommender systems, you also specify offline model property values within the Model Reader window itself. For more information, see "Using Recommender Systems" in *SAS Event Stream Processing: Using Streaming Analytics*.

For ONNX models, you also specify offline model property values within the Model Reader window but do not receive request events.

For ONNX models, you can also specify pre-processing and warm-up parameters within the Model Reader window.

For more information, see "Working with ONNX Models" in *SAS Event Stream Processing: Using Streaming Analytics*.

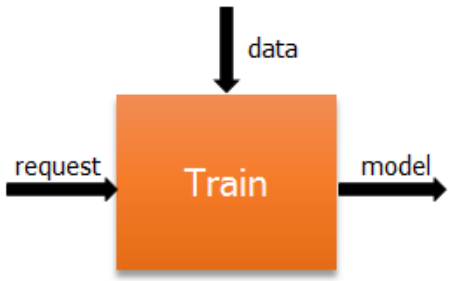
Table 5.2 *Properties of the window-model-reader Element*

Property	Required or Optional	Description
`name`	Required	Specifies the window name. It must start with one of the following characters: _, a–z, A–Z. The rest of the name can include the following characters: _, a–z, A–Z, 0–9.
`model-type`	Optional	Specifies the type of model to read and pass to a Score window. Valid options are **`astore`** and **`recommender`** for analytic store and Recommender System offline models, respectively.
`pubsub`	Optional	Specifies whether to publish and subscribe to the window. When the project-level value of `pubsub` is **`manual`**, a value of **`true`** enables publishing and subscribing and a value of **`false`** disables it.

Using Train Windows

Train windows receive data events and publish model events to Score windows. They use incoming data events to develop and adjust model parameters in real time. Often, the data is historical data from which to learn patterns. Incoming data should contain both the outcome that you are trying to predict and related variables.

Train windows can also receive request events. These events can adjust the learning algorithm while events continue to stream.



After a Train window has adjusted an algorithm, it writes the adjusted model to a Score window or a Model Supervisor window through a model event.

Table 5.3 *Properties of the window-train element*

Property	Required or Optional	Description
`name`	Required	Specifies the window name. It must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
`algorithm`	Required	Specifies the algorithm for the window. If the project is online, the algorithm must be specified with its short name.
`pubsub`	Optional	Specifies whether to publish and subscribe to the window. When the project-level value of `pubsub` is **`manual`**, a value of **`true`** enables publishing and subscribing and a value of **`false`** disables it.

The following is a generic example of a Train window element:

```
<window-train name='w_train' algorithm='algorithm_short_name'>
        <parameters>
```

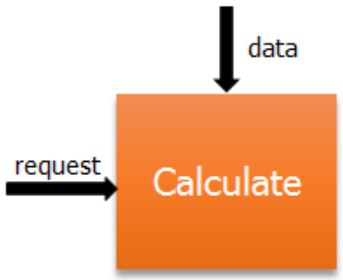
```
    <properties>
     ...
    </properties>
  </parameters>
  <input-map>
   ...
  </input-map>
</window-train>
```

For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

Using Calculate Windows

Overview to Calculate Windows

Calculate windows create real time, running statistics that are based on established analytical techniques. They receive data events and publish newly transformed score data into output events. (No role is assigned to the outgoing edges, so those edges do not appear in the diagram.) Calculate windows can also receive request events.



Calculate windows are designed for data normalization and transformation methods, as well as for learning models that bundle training and scoring together.

Table 5.4 *Properties of window-calculate element*

Property	Required or Optional?	Description
`name`	Required	Specifies the window name. It must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.

Property	Required or Optional?	Description
`algorithm`	Required	Specifies the algorithm for the window. If the project is online, the algorithm must be specified with its short name.
`collapse-updates`	Optional	If **`true`**, multiple update blocks are collapsed into a single update block.
`exp-max-string`	Optional	Specifies the maximum size of strings that the expression engine uses for the window. The default value is **`1024`**.
`index`	Optional	Specifies the index type for the window. Valid options are **`'pi_RBTREE'`**, **`'pi_HASH'`**, **`'pi_LN_HASH'`**, **`'pi_CL_HASH'`**, **`'pi_FW_HASH'`**, **`'pi_EMPTY'`**, **`'pi_HLEVELDB'`**, or **`'pi_HLEVELDB_NC'`**.
`output-insert-only`	Optional	If **`true`**, prevents the window from passing non-insert events to other windows.
`pubsub`	Optional	Specifies whether to publish and subscribe to the window. When the project-level value of `pubsub` is **`manual`**, a value of **`true`** enables publishing and subscribing and a value of **`false`** disables it.
`pubsub-index`	Optional	Specifies the publish/subscribe index value. Valid options are **`'pi_RBTREE'`**, **`'pi_HASH'`**, **`'pi_LN_HASH'`**, **`'pi_CL_HASH'`**, **`'pi_FW_HASH'`**, **`'pi_EMPTY'`**,

Property	Required or Optional?	Description
		'pi_HLEVELDB', or **'pi_HLEVELDB_NC'**.
`pulse-interval='interval (unit)'`	Optional	Specifies the interval at which to write a canonical batch of updates. Pass an integer for the `interval`. Valid options for the `unit` are **microseconds**, **millseconds**, **seconds**, **minutes**, **hours**, or **days**. The default is **milliseconds**.

The following is a generic example of a Calculate window element:

```
<window-calculate name='w_calculate' algorithm='algorithm_short_name'>
          <schema>
               <fields>
                  ...
               </fields>
            </schema>
            <parameters>
              <properties>
                ...
              </properties>
            </parameters>
            <input-map>
             ...
            </input-map>
            <output-map>
              ...
            </output-map>
            <connectors>
            ...
            </connectors>
          </window-calculate>
```

You can define a SAS Micro Analytic Service module to specify a block of code to execute within a Calculate window. The block of code can contain one or more functions. You can write this block of code in Python or DS2. For more information about coding in Python, see "Python Support in SAS Micro Analytic Service" in *SAS Micro Analytic Service: Programming and Administration Guide*. For more information about coding in DS2, see "DS2 Programming for SAS Micro Analytic Service" in *SAS Micro Analytic Service: Programming and Administration Guide*.

You must specify a SAS Micro Analytic Service map within the Calculate window to bind functions to a Source window, which receives the input data.

Working with SAS Micro Analytic Service Modules

Overview

SAS Micro Analytic Service is a memory-resident, high-performance program execution service. A SAS Micro Analytic Service module is essentially a named block of code that you execute within a SAS Event Stream Processing project. This block of code can contain one or more functions that can be written in Python or DS2. You define a module at the project level.

Using Python within SAS Micro Analytic Modules

Environment Variables

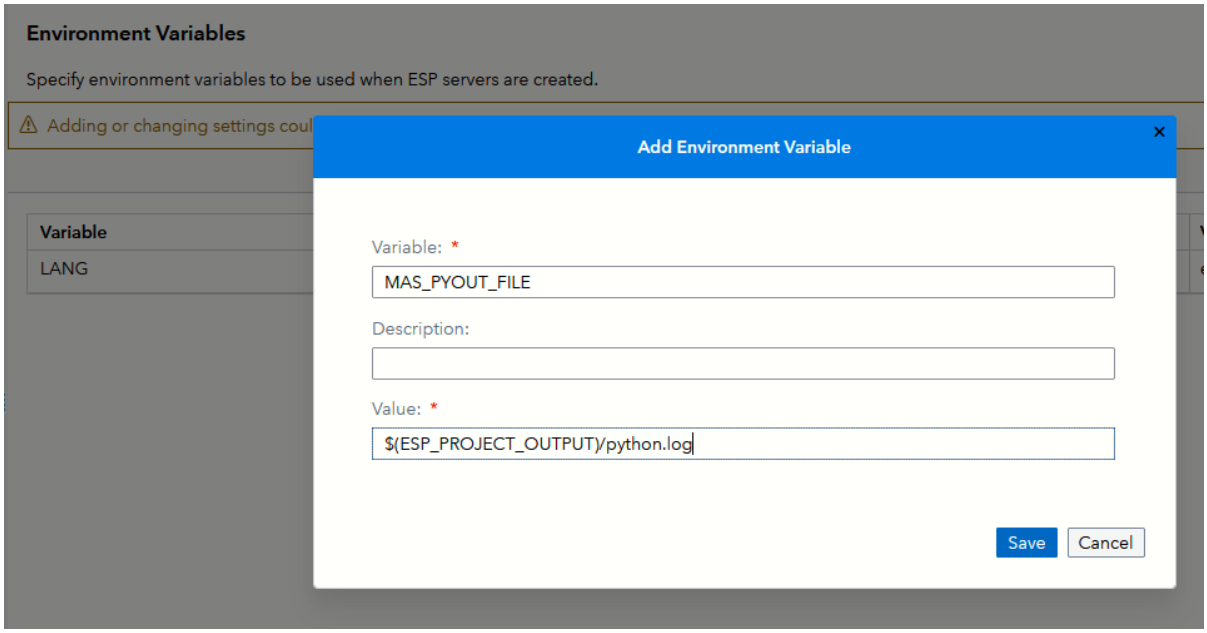
In order to use Python functions within SAS Micro Analytic modules, two environment variables are set:

- MAS_M2PATH specifies the full path name to a required Python program, `mas2py.py`. Regardless of whether you are running SAS Event Stream Processing in a Kubernetes environment or on-premises, you do not need to change the value of this environment variable.
- MAS_PYPATH specifies the full path name to the Python executable. When SAS Event Stream Processing runs in a Kubernetes environment, this name points to an embedded version of Python. When you are running the product on-premises, set it to the location of an installed version of Python (3.x). For example, set `MAS_PYPATH=/usr/bin/python`.

Within a SAS Micro Analytic module, the messages that your Python code writes to standard output (`stdout`) and standard error (`stderr`) are not preserved. To save these messages to an external file, you need to set the MAS_PYOUT_FILE environment variable. Specify the value of this variable as a full path and file name.

By default, this file is overwritten each time a project restarts. In order to append to the file, add `+` to the value that you specify for MAS_PYOUT_FILE (for example, `MAS_PYOUT_FILE="+/mnt/data/project/output/python.log"`).

In a Kubernetes environment, set the value of MAS_PYOUT_FILE on the **User Session** section of the **Settings** page. For example:



Note: The previous figure assumes the use of a project package. For more information, see "Project Package" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

In an edge environment, set the value of MAS_PYOUT_FILE on the system console.

```
export MAS_PYOUT_FILE=/directory/filename
```

Adding Packages to Python

If your SAS Micro Analytic modules that run Python code require additional Python packages, then use the SAS Configurator for Open Source (`sas-pyconfig`) utility to download, configure, and build a custom installation. For more information, see "Configure Open Source Integration Points" in *SAS Viya Platform: Deployment Guide*.

After you run this utility, you have a new Python installation within a Persistent Volume Claim (PVC). To use this new installation, add the `sas-pyconfig` PVC volume to the template for the `ESPConfig` element in `espconfig-sas-pyconfig.yaml`.

Here is an example:

```
apiVersion: builtin
kind: PatchTransformer
metadata:
  name: patch-transformer-esp-pyconfig
patch: |-
  - op: add
    path: /spec/projectTemplate/deployment/spec/template/spec/volumes/-
    value:
       name: python-volume
       persistentVolumeClaim:
          claimName: sas-pyconfig
  - op: add
```

```
    path: /spec/projectTemplate/deployment/spec/template/spec/
containers/0/volumeMounts/-
    value:
       name: python-volume
       mountPath: /opt/sas/viya/home/sas-pyconfig
       readOnly: true
target:
  group: iot.sas.com
  kind: ESPConfig
  name: sas-esp-operator
  version: v1
```

Put this patch in `site-config` and add a line to `kustomization.yaml` in the `transformers` section:

```
site-config/espconfig-sas-pyconfig.yaml
```

Update the MAS_PYPATH variable in SAS Event Stream Processing Studio and SAS Event Steam Manager to point to this Python instance. Based on the preceding example, the value should be **`/opt/sas/viya/home/sas-pyconfig/default_py/bin/python3`**.

For information about how to update MAS_PYPATH in SAS Event Stream Processing Studio, see "Specify Environment Variables and ESP Server Properties for All Users" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

For information about how to update it in SAS Event Steam Manager, see "Global Settings" in *SAS Event Stream Processing: Using SAS Event Stream Manager*.

To add additional Python libraries later, rerun `sas-pyconfig`. You can also add additional profiles. You need to update MAS_PYPATH to point to the profile of the Python installation that you want to run.

Defining a SAS Micro Analytic Service Map

You define a SAS Micro Analytic Service map in a Calculate window to bind a function to a Source window. This binding acts as the input handler for the Source window.

Consider the following example that calculates the total cost of a stock transaction:

```
<project name="MASPython" threads="1" pubsub="auto" heartbeat-
interval="1" luaroot="@ESP_PROJECT_OUTPUT@/luaroot">
  <mas-modules>
    <mas-module module="module1" language="python" func-
names="compTotal"> <!-- 1 -->
      <code><![CDATA[def compTotal(quantity, price):<!-- 2 -->
    "Output: total"
    total= quantity * price
    return total]]></code>
    </mas-module>
  </mas-modules>
  <contqueries>
    <contquery name="cq1">
```

```
      <windows>
        <window-source name="src_win">
          <schema>
            <fields>
              <field name="ID" type="string" key="true"/>
              <field name="security" type="string"/>
              <field name="quantity" type="double"/>
              <field name="price" type="double"/>
            </fields>
          </schema>
          <connectors>
            <connector name="New_Publisher_1" class="fs">
              <properties>
                <property name="type"><![CDATA[pub]]></property>
                <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/
files/python_trades.csv]]></property>
                <property name="fstype"><![CDATA[csv]]></property>
              </properties>
            </connector>
          </connectors>
        </window-source>
        <window-calculate name="pw1" algorithm="MAS">
          <schema> <!--3-->
            <fields>
              <field name="ID" type="string" key="true"/>
              <field name="security" type="string" key="false"/>
              <field name="quantity" type="double" key="false"/>
              <field name="price" type="double" key="false"/>
              <field name="total" type="double" key="false"/>
            </fields>
          </schema>
          <mas-map>
            <window-map module="module1" function="compTotal"
revision="0" source="src_win"/><!--4-->
          </mas-map>
          <connectors>
            <connector name="New_Subscriber_1" class="fs">
              <properties>
                <property name="type"><![CDATA[sub]]></property>
                <property name="snapshot"><![CDATA[false]]></property>
                <property name="fsname"><![CDATA[@ESP_PROJECT_OUTPUT@/
output.csv]]></property>
                <property name="fstype"><![CDATA[csv]]></property>
              </properties>
            </connector>
          </connectors>
        </window-calculate>
      </windows>
      <edges>
        <edge source="src_win" target="pw1" role="data"/><!--5-->
      </edges>
    </contquery>
  </contqueries>
</project>
```

1. SAS Micro Analytic Service module named `module1` is defined at the project level of the `MASPython` project. It contains a function `compTotal` written in Python.
2. The Python code for `compTotal` is specified within the `<code>` element. This function acts as the input handler for all events passed from the `src_win` Source window to the `pw1` Calculate window.
3. The schema specifies the fields that define the structure of incoming events. Data relevant to the security being traded, the quantity of shares, and the current prices are streamed into the Calculate window. The schema also specifies the total that is calculated by the `compTotal` fuction.
4. At the map level of the Calculate window, the window map binds the `compTotal` function of `module1` to the Source window.
5. The edge between the Source window and the Calculate window must have the role `data`.

Note: The previous code block references input and output files in the directories of a project package. For more information, see "Project Package" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

A Calculate window can have only one SAS Micro Analytic Service module map. This map can include one or more window maps.

For example, in the following code, two SAS Micro Analytic Service modules containing DS2 functions are defined at the project level:

```
<mas-modules>
   <mas-module language="ds2" module="module_1" func-
names='compute_volume'>
       <code>
         <![CDATA[
               ds2_options sas;
               package myModule_1/overwrite=yes;
               method compute_volume(int quantity, double price,
in_out int volume);
               volume = quantity * price;
               end;
               endpackage;
          ]]>
       </code>
    </mas-module>
    <mas-module language="ds2" module="module_2" func-
names='compute_price_sqr'>
       <code>
         <![CDATA[
               ds2_options sas;
               package myModule_2/overwrite=yes;
               method compute_price_sqr(int quantity, double price,
in_out double price_sqr);
               price_sqr = price * price;
               end;
               endpackage;
          ]]>
       </code>
```

```
    </mas-module>
</mas-modules>
```

The ESP server loads the modules in the order in which they appear in the XML code.

Two window maps within a SAS Micro Analytic Service map bind the DS2 functions to the Calculate window:

```
...
<mas-map>
   <window-map module="module_1" revision="0" source="Trades1"
function="compute_volume"/>
   <window-map module="module_2" revision="0" source="Trades2"
function="compute_price_sqr"/>
</mas-map>
...
```

Window maps have four attributes:

- `module` refers to the name of a previously defined module
- `revision` must be set to 0
- `source` is the name of the Source window for which you specify the handler
- `function` specifies the function defined in the module

For information about the XML elements that you use to define a SAS Micro Analytic Service module, see "XML Language Elements Relevant to Streaming Analytics Windows" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Generating Multiple Derived Events

A SAS Micro Analytic Service module method can write one or more arrays. You can generate multiple derived events from a single input event when at least one of the output array names matches a derived event field name.

The number of events generated is determined by the number of elements in the matching arrays at run time.

SAS Event Stream Processing requires every insert event to have a unique key. Therefore, a source event's key cannot be duplicated in multiple generated derived events. When the key is duplicated, SAS Event Stream Processing halts processing and returns an error.

When you map a SAS Micro Analytic Service module function to a Calculate window, ensure that each derived event has a unique key value.

Consider the following input event:

```
i,n,1,field1,field2,field3
```

Assume that the Source window of the project uses the following input schema:

```
<schema>
    <fields>
      <field name="ID" type="int32" key="true"/>
```

```
        <field name="field1" type="string"/>
        <field name="field2" type="string"/>
        <field name="field3" type="string"/>
      </fields>
</schema>
```

And that the Calculate window has the following output schema:

```
<schema>
    <fields>
       <field name="ID" type="int32" key="true"/>
       <field name="_masRowNum" type="int32" key="true"/>
       <field name="stringD" type="string"/>
    </fields>
</schema>
```

Assume that the Calculate window uses the following SAS Micro Analytic Service map:

```
<mas-map>
     <window-map module="masds2" revision="0" source="Src"
function="transpose"/>
 </mas-map>
```

You can use the following SAS Micro Analytic Service module to transpose the input event:

```
<mas-modules>
  <mas-module language="ds2" module="masds2" func-names='transpose'>
    <code>
      <![CDATA[
             ds2_options sas;
             package test_package;
               method transpose(nchar(50) field1, nchar(50) field2,
                                nchar(50) field3, in_out nchar(50)
                                stringD[3]);
                 stringD[1] = trim(field1);
                 stringD[2] = trim(field2);
                 stringD[3] = trim(field3);
               end;
             endpackage;
          ]]>
    </code>
  </mas-module>
</mas-modules>
```

The resulting output events would be as follows:

```
i,n,1,1,field1
i,n,1,2,field2
i,n,1,3,field3
```

The combination of `ID` from the Source window and `_masRowNum` in the Calculate window is the key in the Calculate window output schema. Also note that the field named `stringD` in the Calculate window output schema matches the array name `stringD[]` in the DS2 code as is required.

For more information, see "Unique Keys" in *SAS Micro Analytic Service: Programming and Administration Guide*.

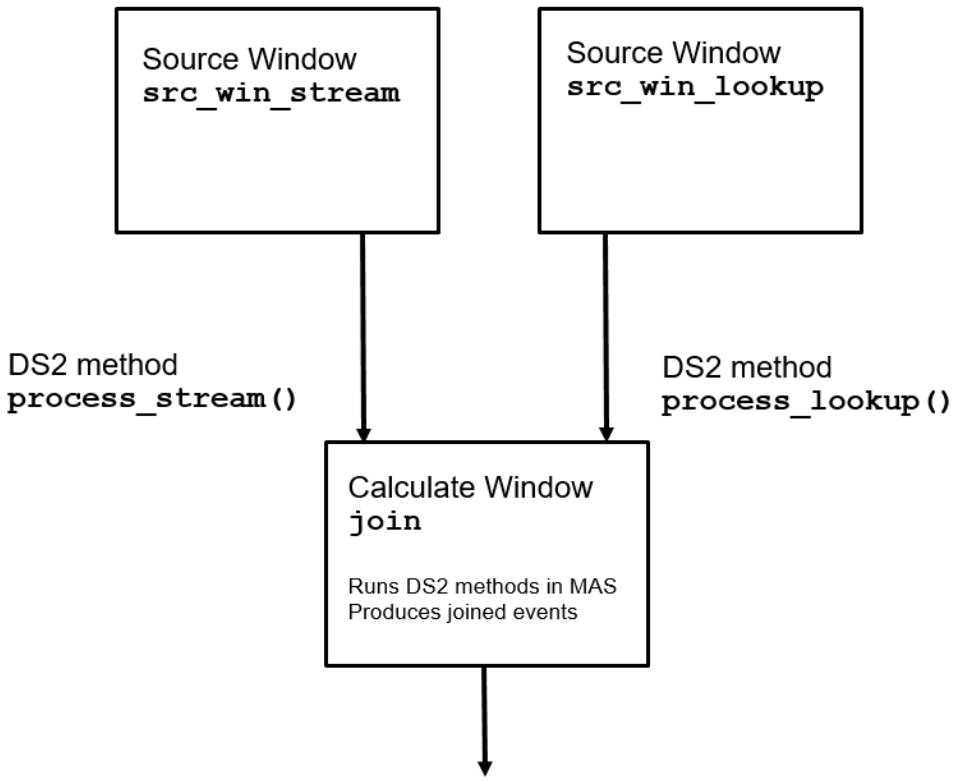
Using Shared Vectors and Shared Hash Table Data Structures

SAS Micro Analytic Service provides two ways to share data between the modules executing within a user context: shared vectors and shared hash tables. *Shared vectors* are collections of data values. The values within a vector can have a mix of data types. *Shared hash tables* are containers of vectors, where the vectors are stored and accessed by using keys. Both of these structures are thread–safe and lock-free. You can use them to share data across DS2 methods within a SAS Event Stream Processing project.

For more information about these data structures, see "State Sharing between Modules" in *SAS Micro Analytic Service: Programming and Administration Guide*.

The following query shows two Source windows streaming events into a Calculate window. Two DS2 methods that use a shared hash table run within a SAS Micro Analytic Service module. The module simulates a no-regenerate inner-join. The module runs within a Calculate window and produces joined events.

Figure 5.1 *Continuous Query with a SAS Micro Analytic Service Module Sharing a Hash Table*



Here is the code for the SAS Micro Analytic Service module:

```
<mas-modules>
   <mas-module language="ds2" module="module_1"
```

```
                func-names='process_lookup,process_stream'>
      <code>
        <![CDATA[
           ds2_options sas;
             package module_1/overwrite=yes;

             dcl package masstate /*1*/ st();
             dcl package logger logr('DF.ESP');

             method process_stream /*2*/(varchar(16) _inOpcode, bigint
lookupID,
                    in_out varchar matchString,
                    in_out varchar matchDescription,
                    in_out varchar _outOpcode) ;
                dcl int rc;
                dcl varchar(32) key;

                key = lookupID;
                _outOpcode='noop'; /*3*/

                /* logr.log('d', 'key=$s', key); */
                if (0 eq st.get(key)) then do;
                    matchString=st.getString(key,0);
                    matchDescription=st.getString(key,1);
                    _outOpcode=_inOpcode; /*4*/
                end;
                /* logr.log('d', 'inOpcode=$s, outOpcode=$s',
_inOpcode, _outOpcode); *//*5*/
             end;

             method process_lookup /*6*/(bigint ID, varchar(32) value,
                    varchar(4096) description,
                    in_out varchar _outOpcode);
                dcl int rc;
                dcl varchar(32) key;

                _outOpcode='noop';
                key = ID;
                rc = st.createVector(key,2);  /*7*/
                rc = st.setString(key, 0, value); /*8*/
                rc = st.setString(key, 1, description);
                rc = st.put(key); /*9*/

                rc = st.deleteVector(key); /*10*/
             end;
           endpackage;
           ]]>
         </code>
    </mas-module>
</mas-modules>
```

1 When using a shared hash table, you use the package MASSTATE to create, share, retrieve, and delete data.

2 The method `process_stream` is designed to insert received events into a shared hash table. It can also handle Update and Delete events by looking at the `_inOpcode`.

3 Default to **noop**. That is, produce no event.

4 Set the `Opcode` upon a match, producing an event.

5 Uncomment this line to write debugging messages to the log.

6 The method `process_lookup` is designed to look up a string through a lookup `ID` in a shared hash table. If no match is found, then the `_outOpcode` is **noop**, so no event is generated.

The method as presented handles Insert events. You could modify it to examine the `_inOpcode` and add, update, or delete an entry from the shared hash table.

7 Create a new vector.

8 Insert the events data.

9 Put the vector to the hash.

10 Delete the vector.

Here is the XML code for two Source windows:

```
<window-source name='src_win_lookup' index='pi_RBTREE'>
      <schema-string>ID*:int64,value:string,description:string</schema-
string>
      <connectors>
         <connector class='fs' name='pub'>
            <properties>
               <property name='type'>pub</property>
               <property name='fstype'>csv</property>
               <property name='fsname'>lookup.csv</property>
            </properties>
         </connector>
      </connectors>
</window-source>

<window-source name='src_win_stream' index='pi_RBTREE'>
    <schema-string>ID*:int64, lookupID:int64, price:double,
quant:int64</schema-string>
    <connectors>
       <connector class='fs' name='pub'>
         <properties>
           <property name='type'>pub</property>
           <property name='fstype'>csv</property>
           <property name='fsname'>stream.csv</property>
         </properties>
       </connector>
    </connectors>
</window-source>
```

Here is the XML code for the Calculate window containing the SAS Micro Analytic Service map for the methods:

```
<window-calculate name='join' algorithm='MAS'>
    <schema-
string>ID*:int64,matchID:int64,matchString:string,matchDescription:stri
ng,
                    price:double,quant:int64</schema-string>
    <mas-map>
```

```
      <window-map module="module_1" revision="0"
source="src_win_lookup" function="process_lookup"/>
      <window-map module="module_1" revision="0"
source="src_win_stream" function="process_stream"/>
    </mas-map>
    <connectors>
      <connector class='fs' name='sub'>
        <properties>
          <property name='type'>sub</property>
          <property name='fstype'>csv</property>
          <property name='fsname'>result.csv</property>
          <property name='snapshot'>true</property>
        </properties>
      </connector>
    </connectors>
</window-calculate>
```

Here is the XML code for the edges between the Source windows and the Calculate window:

```
<edges>
   <edge source='src_win_lookup' target='join' role='data'/>
   <edge source='src_win_stream' target='join' role='data'/>
</edges>
```

Here is the XML code for the connector groups. You place this code outside the continuous query that contains the Source and Calculate windows.

```
<project-connectors>
      <connector-groups>
         <connector-group name="group1"> <!--1-->
            <connector-entry connector='cq_01/join/sub'
state='running'/>
            <connector-entry connector='cq_01/src_win_lookup/pub'
state='finished'/>
         </connector-group>
         <connector-group name="group2">
            <connector-entry connector='cq_01/src_win_stream/pub'
state='finished'/>
         </connector-group>
      </connector-groups>
      <edges>
         <edge source='group1' target='group2'/>
      </edges>
   </project-connectors>
```

1 A *connector group* is a container of connector-entry elements. Connector entries can specify the state of a connector.

Now suppose that you stream the following events into the Source window src_win_lookup:

```
i,n,10001,sunw,"Workstation manufacturer"
i,n,20001,ibm,"From typewriters to mainframes"
```

Then you stream the following events into the Source window src_win_stream:

```
i,n,1,10001,101.45,100
i,n,2,20001,23.5,1000
i,n,3,20001,10.1,80
```

```
i,n,4,10001,10.2,85
```

The joined data in XML is as follows:

```
<event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>1</value>
   <value name='matchDescription'>Workstation manufacturer</value>
   <value name='matchString'>sunw</value>
   <value name='price'>101.450000</value>
   <value name='quant'>100</value>
 </event>

 <event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>2</value>
   <value name='matchDescription'>From typewriters to mainframes</value>
   <value name='matchString'>ibm</value>
   <value name='price'>23.500000</value>
   <value name='quant'>1000</value>
 </event>

 <event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>3</value>
   <value name='matchDescription'>From typewriters to mainframes</value>
   <value name='matchString'>ibm</value>
   <value name='price'>10.100000</value>
   <value name='quant'>80</value>
 </event>

 <event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>4</value>
   <value name='matchDescription'>Workstation manufacturer</value>
   <value name='matchString'>sunw</value>
   <value name='price'>10.200000</value>
   <value name='quant'>85</value>
 </event>
```

Using Score Windows

Score windows accept model events to make predictions for incoming data events. They generate scored data. You can use this score data to generate predictions based on the trained model. (No role is assigned to the outgoing edges, so they do not appear in the diagram.)

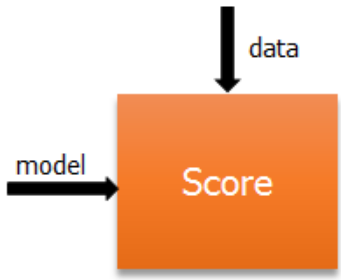


Table 5.5 *Properties of window-score element*

Property	Required or Optional	Description
`name`	Required	Specifies the window name. It must start with one of the following characters: _, a-z, A-Z. The rest of the name can include the following characters: _, a-z, A-Z, 0-9.
`model-type`	Optional	Specifies the type of model to read and pass to a Score window. Valid options are **`astore`** and **`recommender`**, for analytic store and Recommender System offline models, respectively.
`pubsub`	Optional	Specifies whether to publish and subscribe to the window. When the project-level value of `pubsub` is **`manual`**, a value of **`true`** enables publishing and subscribing and a value of **`false`** disables it.

The following is a generic example of a Score window element:

```
<window-score name='w_score'>
          <schema>
            <fields>
              ...
            </fields>
          </schema>
          <models>
           ...
          </models>
        </window-score>
```

For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

6

Custom Windows

Working with Custom Windows

Overview

Custom windows enable an end user to reuse external code across projects. First, a developer creates a configuration file that uses Python or Lua code to create a custom window. Then, someone uses SAS Event Stream Processing Studio to make it available to end users.

The configuration file specifies both the configuration of the custom window and the actions that are performed by the window on incoming events. The code in the file can access all the common server features that are available to both Python and Lua. A `create` function must be specified within the code to generate output events.

Event streams from an input window are mapped to properties that are specified in the custom window. The custom window uses the values of those properties as it calls the `create` function. The code returns one or more objects to the custom window, which then maps fields from an object to event fields of output events.

This topic explains how developers can create the code for a custom window. For more information about how developers manage custom windows and how users can use them, see "Working with Custom Windows" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

Visit SAS Event Stream Processing Studio Custom Windows GitHub for a custom window example and to obtain custom windows that you can add to SAS Event Stream Processing Studio.

Creating the Configuration File

Overview

The code that specifies the actions that a custom window performs on incoming events references four functions used by the ESP server:

- `create` - a function called to generate one or more events.

  IMPORTANT This function is required. If your code does not contain this function, then the ESP server returns an error.

- `init` - an optional function that initializes program values. It is called after the window is created.
- `heartbeat` - an optional function called at the project heartbeat interval.
- `destroy` - an optional function called when the window is destroyed.

The code must contain a global variable named `_espconfig_` that defines the window configuration. The `_espconfig_` global variable is specified as either a Python dictionary or a Lua table.

Processing Incoming and Outgoing Events

The code specifies the functional entry points that are called at appropriate times during the event streaming process. The only required function is the `create` function.

The `init` function is called when the window is created. It is passed any initialization data that is defined in the configuration file. The function signature looks like this (Python and Lua):

```
def init(settings):
function init(settings)
```

The `create` function is called when the window receives an input event. This function signature can be in one of two formats, depending on whether the parameters are expanded.

- If the value of `expand-parms` is **`false`**, the signature looks like this:

```
def create(data,context):
function create(data,context)
```

Here, `data` is an object that contains the fields that were defined in the `inputVariables` section of the configuration file. The `context` parameter is an object that contains the name of the custom window and the name of the input window.

- If the value of `expand-parms` is **`true`**, the signature contains a parameter for each entry in the `inputVariables` section of the configuration file.

Suppose you process the following input variables:

```
"inputVariables": {
    "desc" : "Optional description of the input variables",
    "fields" : [
        {
            "name": "trade_symbol",
            "desc": "Stock symbol"
        },
        {
            "name": "trade_price",
            "desc": "Trade price"
        },
        {
            "name": "trade_quantity",
            "desc": "Trade quantity"
        }
    ],
}
```

In that case, the function signature would be as follows:

```
def create(symbol,price,quant):
function create(symbol,price,quant)
```

The sequence is always the order in which the inputs are defined in the configuration. Note that the names do not need to match the names in the configuration.

The `create` function returns 0 or more objects that contain field names in the `outputVariables` section of the configuration file.

This example shows output variables dealing with stock trade data:

```
"outputVariables": {
    "desc" : "Optional description of the output variables",
    "fields" : [
        {
            "name": "maxp",
            "desc": "Maximum price"
        },
        {
            "name": "maxq",
            "desc": "Maximum quantity"
        }
    ],
}


def create(symbol,price,quant):

    ...
```

```
    event = {}

    event["symbol"] = symbol
    event["maxp"] = maxprice
    event["maxq"] = maxquant

    return event
```

If you return a null value from the `create` function, no events are generated. If you return a list of objects, an event is generated for each item in the list.

The custom window maps any value that is defined in the `outputVariables` section of the configuration file to the event field that is specified in the `<plugin>` element. In addition, the custom window copies any fields that it has in common with the input window into the new event.

The `heartbeat` function is called when the custom window receives a heartbeat from the server. This happens at the interval that was defined for the project.

The signature is as follows:

```
def heartbeat(context):
function heartbeat(context)
```

The `context` parameter is an object that contains the name of the custom window.

The `destroy` function is called when the custom window is destroyed in the server.

The signature is as follows:

```
def destroy():
function destroy()
```

The Lua or Python code can throw exceptions. Exceptions from the `init` function cause the project to fail to load. Exceptions from the `create` function, the `heartbeat` function, and snippets (when threaded) are handled based on the `on-script-exception` attribute. For more information, see "Defining a Custom Window".

Specifying the Window Configuration Object

Specify the window configuration object as a variable named `_espconfig_`. Use either a Python dictionary or a Lua table. The language that you use for this object must be the same as the language of the code to be executed by the window.

`_espconfig_` contains the following entries:

- `settings` - an object that contains operational settings for the custom window. Supported properties are as follows:
  - `desc` - an optional description of the settings.
  - `expand-parms` - determines whether the `create` function receives individual parameters for each entry in the `inputVariable` section or an object that contains fields that map to the names of the input variables. This property defaults to **`false`**.

Here is an example of a `settings` entry in which the property is false:

```
"settings" : {
    "expand-parms":false
}
```

The `create` function signature would be as follows, assuming that the input variables are named a, b, and c:

```
def create(data,context):
    aVal = data["a"]
    bVal = data["b"]
    cVal = data["c"]
    ...
```

Here is an example, of a `settings` entry in which the property is true:

```
"settings" : {
    "expand-parms":true
}
```

The `create` function signature would be as follows:

```
def create(aVal,bVal,cVal):
    ...
```

- `process-blocks` - determines whether the `create` function receives events in blocks that contain multiple events or a single event at a time. The property defaults to **`false`**.

  Note: If `process-blocks` is **`true`** and `expand-parms` is **`true`**, an error is returned.

- `encode-binary` - determines whether binary data that is passed into the `create` function is Base64 encoded. The property defaults to **`false`**.

- `inputVariables` - an array that contains objects that map input data from an incoming window to the custom window's `create` function. Each object contains the following fields:
  - `fields` - contains the fields that are to be mapped.
  - `name` - the name used to populate the object that is passed into the `create` function (when parameters are not expanded).
  - `desc` - an optional description of the field.
  - `esp_type` - specifies the data type of the fields that the user of the custom window is permitted to map to input variables.
  - `optional` - an optional Boolean value that indicates whether the field is optional. When **`true`**, a mapping entry for the input field is not required for the project.

  The following example shows an `inputVariables` array for an ESP project that processes stock trade data. The custom window is processing trade data by symbol:

```
"inputVariables" : {
        "desc" : "Variables from the input event",
        "fields" : [
```

```
            {
                "name": "trade_symbol",
                "desc": "Stock symbol",
                "esp_type": "string",
                "optional": False
            },
            {
                "name": "trade_price",
                "desc": "Trade price",
                "esp_type": "int64",
                "optional": False
            },
            {
                "name": "trade_quantity",
                "desc": "Trade quantity",
                "esp_type": "int64",
                "optional": False
            }
        ]
    }
```

- `outputVariables` - an array that contains objects to map output data from the `create` function to event fields in the output. Each object has the following fields:
  - `fields` - contains the fields that are to be mapped.
  - `name` - the name used to populate the object that is passed into the `create` function (when parameters are not expanded).
  - `desc` - an optional description of the field.
  - `esp_type` - specifies the data type of the fields that the user of the custom window is permitted to map to output variables.

  The following example shows an `outputVariables` array for a custom window that tracks the maximum price and quantity of a trade by symbol:

```
"outputVariables" : {
        "desc" : "Variables created for the output event",
        "fields" : [
            {
                "name": "maxp",
                "desc": "Maximum price",
                "esp_type": "int64"
            },
            {
                "name": "maxq",
                "desc": "Maximum quantity",
                "esp_type": "int64"
            }
        ]
    }
```

  The `create` function would return one or more objects with `maxp` and `maxq` fields. The custom window uses the `<output-map>` to map these fields to the appropriate output event field.

```
<output-map>
  <properties>
```

```
    <property name="maxp" esp_type="int64">max_price</property>
    <property name="maxq" esp_type="int64">max_quant</property>
  </properties></output-map>
```

In this case, the custom window sets the value of `max_price` to `maxp` and `max_quant` to `maxq` and specifies that data type must be `int64`. All other fields that are common to the input window and the custom window are copied as is.

- `initialization` - an array that contains objects that are passed into the `init` function. The custom window puts the values into a single data object, which is passed into the function. If this array exists in the configuration, then you must provide an `init` function or you get an error.

  Each object has the following fields:

  - `fields` - contains the fields that are to be mapped.
  - `name` - the name of the property.
  - `desc` - an optional description of the property.
  - `default` - an optional default value of the property. This value is used when the model does not provide a value in the `<plugin>` element. If no default value is specified and the `<plugin>` element does not contain an entry, the property is not sent to the `init` function.
  - `input_type` - specifies that initialization values are displayed to the user of the custom window as a drop-down list. When `input_type = "dropdown"` is present, a drop-down list is used. When `input_type` is not present, a text box is used.
  - `values` - specifies a comma-separated list of initialization values that the user of the custom window can choose from.

  The following example shows an `initialization` array for a custom window that tracks the maximum price and quantity of a trade by symbol:

```
"initialization" : {
        "desc" : "Some initialization values",
        "fields" : [
            {
                "name": "symbols",
                "desc": "",
                "default": "AAPL",
                "input_type": "dropdown",
                "values": ["AAPL","IBM","T"]
            }
        ]
    }
```

Here is a sample `_espconfig_` written in Python:

```
_espconfig_ = {
    "settings" : {
        "desc" : "Some settings for the custom window",
        "expand_parms" : True,
        "process_blocks" : False,
        "encode_binary" : False
    },
    "inputVariables" : {
        "desc" : "Variables from the input event",
```

```
            "fields" : [
                {
                    "name": "trade_symbol",
                    "desc": "Stock symbol",
                    "esp_type": "string",
                    "optional": False
                },
                {
                    "name": "trade_price",
                    "desc": "Trade price",
                    "esp_type": "int64",
                    "optional": False
                },
                {
                    "name": "trade_quantity",
                    "desc": "Trade quantity",
                    "esp_type": "int64",
                    "optional": False
                }
            ]
        },
        "outputVariables" : {
            "desc" : "Variables created for the output event",
            "fields" : [
                {
                    "name": "maxp",
                    "desc": "Maximum price",
                    "esp_type": "int64"
                },
                {
                    "name": "maxq",
                    "desc": "Maximum quantity",
                    "esp_type": "int64"
                }
            ]
        },
        "initialization" : {
            "desc" : "Some initialization values",
            "fields" : [
                {
                    "name": "symbols",
                    "desc": "",
                    "default": "AAPL",
                    "input_type": "dropdown",
                    "values": ["AAPL","IBM","T"]
                }
            ]
        }
    }
```

Here is a sample _espconfig_ written in Lua:

```
    espconfig = {
        settings = {
            desc = "Some settings for the custom window",
            expand_parms = true,
            process_blocks = false,
```

```
        encode_binary = false
    },
    inputVariables = {
        desc = "Variables from the input event",
        fields = {
            {
                name = "trade_symbol",
                desc = "Stock symbol",
                esp_type = "string",
                optional = false
            },
            {
                name = "trade_price",
                desc = "Trade price",
                esp_type = "int64",
                optional = false
            },
            {
                name = "trade_quantity",
                desc = "Trade quantity",
                esp_type = "int64",
                optional = false
            }
        }
    },
    outputVariables = {
        desc = "Variables created for the output event",
        fields = {
            {
                name = "maxp",
                desc = "Maximum price",
                esp_type = "int64"
            },
            {
                name = "maxq",
                desc = "Maximum quantity",
                esp_type = "int64"
            }
        }
    },
    initialization = {
        desc = "Some initialization values",
        fields = {
            {
                name = "symbols",
                desc = "",
                default = "AAPL",
                input_type = "dropdown",
                values = ["AAPL", "IBM", "T"]
            }
        }
    }
}
```

Defining a Custom Window

The following XML code shows the general structure of a custom window.

```
<window-custom name="" index="">
  <schema>
    <fields>
      <field name="" type="" key=""/>
      ...
    </fields>
  </schema>
  <plugin code="" on-script-exception="log-and-continue | stop-and-
remove">
    <input-map>
      <properties>
        <property name="" esp_type="">...</property>
        ...
      </properties>
    </input-map>
    <output-map>
      <properties>
        <property name="" esp_type="">...</property>
        ...
      </properties>
    </output-map>
    <initialization>
      <properties>
        <property name="" values="">...</property>
        ...
      </properties>
    </initialization>
  </plugin></window-custom>
```

The top-level element of this code, `<window-custom>`, identifies it as a custom window. A custom window appears in SAS Event Stream Processing Studio with the `name` that you assign. You must specify a `schema` for the custom window.

The `plugin` element contains information about the locations of the code and the mappings of input and output values.

- The `code` attribute specifies the location of the code that is used to support the custom window.
- The `on-script-exception` attribute is an optional attribute that specifies how to handle exceptions that are encountered during the execution of window code. When set to log-and-continue, an error is logged and execution continues. When set to stop-and-remove, a fatal error is logged and the project is stopped and removed. If this value is not specified, the window inherits the script handling behavior defined in the containing project.
- The `input-map` element maps the ESP event variables from the input event to the data that is passed into the `create` function. The `name` attribute contains a name from the `inputVariables` section of the configuration data, and the

content of the element contains the name of the input field. If no content is specified, the `name` value is used. The `esp_type` attribute specifies the data type of the fields that the user of the custom window is permitted to map to input variables.

- The `output-map` element maps the values in the data that is returned from the `create` function to ESP event values in the output event. The `name` attribute contains a name from the `outputVariables` section of the configuration data, and the content of the element contains the name of the output field. If no content is specified, the `name` value is used. The `esp_type` attribute specifies the data type of the fields that the user of the custom window is permitted to map to output variables.
- The optional `initialization` element contains a set of name-value pairs that are passed into the `init` function in the code. The names of the properties must match an entry in the initialization section of the configuration data. The `values` attribute specifies a comma-separated list of initialization values that the user of the custom window can choose from.

Note: When you specify entries in the `initialization` element, you must include an `init` function in your code. Otherwise, an error is generated.

Custom Window Example

The following project processes stock trade data. It uses a custom window that searches for specific stock symbols and calculates the maximum quantity and price for an individual trade. An initialization entry is used to set the ticker symbols of interest.

Here is a configuration file written in Python.

```
symboldata = {}
symbols = None

def init(settings):
    global symbols
    if "symbols" in settings:
        symbols = {}
        for s in settings["symbols"].split(","):
            symbols[s] = True

def create(symbol,price,quant):
    global symboldata

    event = None
    if symbols == None or symbol in symbols:
        if (symbol in symboldata) == False:
            data = {"max_price":price,"max_quant":quant}
            symboldata[symbol] = data
```

```
                event = {}
                event["symbol"] = symbol
                event["maxp"] = price
                event["maxq"] = quant
            else:
                data = symboldata[symbol]

                if (price > data["max_price"] or quant >
data["max_quant"]):
                    if (price > data["max_price"]):
                        data["max_price"] = price
                    if (quant > data["max_quant"]):
                        data["max_quant"] = quant
                    event = {}
                    event["symbol"] = symbol
                    event["maxp"] = data["max_price"]
                    event["maxq"] = data["max_quant"]
                    event["_opcode"] = "upsert"

    return event

_espconfig_ = {
    "settings" : {
        "desc" : "Some settings for the custom window",
        "expand_parms" : True,
        "process_blocks" : False,
        "encode_binary" : False
    },
    "inputVariables" : {
        "desc" : "Variables from the input event",
        "fields" : [
            {
                "name": "trade_symbol",
                "desc": "Stock symbol",
                "esp_type": "string",
                "optional": False
            },
            {
                "name": "trade_price",
                "desc": "Trade price",
                "esp_type": "int64",
                "optional": False
            },
            {
                "name": "trade_quantity",
                "desc": "Trade quantity",
                "esp_type": "int64",
                "optional": False
            }
        ]
    },
    "outputVariables" : {
        "desc" : "Variables created for the output event",
        "fields" : [
            {
                "name": "maxp",
```

```
                "desc": "Maximum price",
                "esp_type": "int64"
            },
            {
                "name": "maxq",
                "desc": "Maximum quantity",
                "esp_type": "int64"
            }
        ]
    },
    "initialization" : {
        "desc" : "Some initialization values",
        "fields" : [
            {
                "name": "symbols",
                "desc": "",
                "default": "AAPL",
                "input_type": "dropdown",
                "values": ["AAPL","IBM","T"]
            }
        ]
    }
}
```

Here is the XML for the project:

```
<project name="p" index="pi_EMPTY" pubsub="none" threads="4">
  <contqueries>
    <contquery name="cq" trace="plugin">
      <windows>
        <window-source name="trades">
          <schema>
            <fields>
               <field name="id" key="true" type="int64"/>
               <field name="symbol" type="string"/>
               <field name="currency" type="int32"/>
               <field name="time" type="int64"/>
               <field name="msecs" type="int32"/>
               <field name="price" type="double"/>
               <field name="quant" type="int32"/>
               <field name="venue" type="int32"/>
               <field name="broker" type="int32"/>
               <field name="buyer" type="int32"/>
               <field name="seller" type="int32"/>
               <field name="buysellflg" type="int32"/>
             </fields>
           </schema>
            <connectors>
              <connector class="fs" active="true">
                <properties>
                  <property name="type">pub</property>
                  <property name="fstype">csv</property>
                  <property name="fsname">input.csv</property>
                </properties>
              </connector>
            </connectors>
          </window-source>
```

```
        <window-custom name="plugin">
          <schema>
            <fields>
              <field name="symbol" type="string" key="true"/>
              <field name="max_price" type="double"/>
              <field name="max_quant" type="int32"/>
            </fields>
          </schema>
          <plugin code="file://symbols.py">
            <input-map>
              <properties>
                <property name="trade_symbol" esp_type="string">symbol
                </property>
                <property name="trade_price" esp_type="double">price
                </property>
                <property name="trade_quantity" esp_type="int32">quant
                </property>
              </properties>
            </input-map>
            <output-map>
              <properties>
                <property name="maxp" esp_type="double">max_price
                </property>
                <property name="maxq" esp_type="int32">max_quant
                </property>
              </properties>
            </output-map>
            <initialization>
              <properties>
                <property name="symbols" values="AAPL,IBM,T">AAPL
                </property>
              </properties>
            </initialization>
          </plugin>
        </window-custom>
      </windows>
      <edges>
        <edge source="trades" target="plugin" />
      </edges>
    </contquery>
  </contqueries></project>
```

7

Advanced Topics

Splitting Generated Events across Output Slots

Overview

Most window types can register a splitter function or expression to determine which output slot or slots should be used for a newly generated event. This capability enables you to send generated events across a set of output slots. By default, windows send events through output slot 0 to zero or more downstream windows.

When you add edges between a window with a splitter function and downstream windows, specify the slot number of the parent window where the downstream window receives its input events.

Using window splitters is more efficient than using two or more subsequent Filter windows. This is because the filtering is performed a single time at the window splitter rather than multiple times for each filter. This results in less data movement and processing.

Using Splitter Functions in Lua

To specify the slot number of an output event, you can create a Lua function that returns an integer that corresponds to the slot number. Consider the following code fragment:

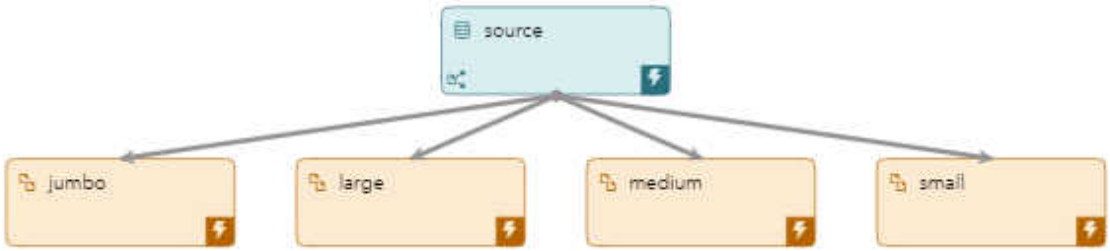
```
<window-...>
    <splitter-lua function="function">
        <use>...</use>
        <code><![CDATA[
            ...
            Lua function that returns an integer = slot number
            ...
        ]]></code>
    </splitter-lua>
</window-...>
```

When the `use` element is specified, the ESP server passes the entire input event or a subset of event fields to the specified function.

Specify the `use` element for performance and efficiency. For example, when you have an input event with 100 fields, but you use only two of them in your Lua code, list those two fields within the `use` element.

Note: Whenever you have a function named `init` in your Lua code, it is called before the model begins to initialize the splitter.

Consider the following project:



Trade data streams to the Source window, which uses slots that are defined in a Lua function named `GetSlot` to send them to one of four Copy windows. The slot is determined by the size of the trade (the `quant` field). The Copy windows are named `small`, `medium`, `large`, and `jumbo` according to the size of the trade.

Note that the code shows the use of a project package to contain the input file.

```
<project name="lua_splitter" index="pi_HASH" pubsub="auto" threads="4">
 <contqueries>
  <contquery name="cq">
   <windows>
    <window-source name="source" insert-only="true">
      <schema>
        <fields>
           <field name="id" type="int64" key="true"/>
           <field name="symbol" type="string" key="false"/>
           <field name="price" type="double" key="false"/>
           <field name="quant" type="int32" key="false"/>
        </fields>
      </schema>
      <connectors>
        <connector class="fs" name="fs">
          <properties>
            <property name="type"><![CDATA[pub]]></property>
            <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/
test_files/trades.csv]]></property>
            <property name="fstype"><![CDATA[csv]]></property>
          </properties>
       </connector>
    </connectors>
    <splitter-lua function="getSlot">
        <use>quant</use>
        <code><![CDATA[
          function getSlot(data)
            if (data.quant < 100) then
              return 0
            elseif (data.quant < 1000) then
              return 1
```

```
            elseif (data.quant < 10000) then
              return 2
            else
              return 3
            end
          end
        ]]></code>
      </splitter-lua>
    </window-source>
    <window-copy name="small">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="medium">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="large">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="jumbo">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
   </windows>
   <edges>
    <edge source="source" target="small" slot="0"/>
    <edge source="source" target="medium" slot="1"/>
    <edge source="source" target="large" slot="2"/>
    <edge source="source" target="jumbo" slot="3"/>
   </edges>
  </contquery>
 </contqueries>
</project>
```

For information about built-in functions that are available to a splitter function in Lua, see Chapter 8, “Common Lua Functionality”.

Using Splitter Functions in Python

To specify the slot number of an output event, you can create a Python function that returns an integer that corresponds to the slot number. Consider the following code fragment:

```
<window-...>
    <splitter-python function="function">
       <use>...</use>
       <code><![CDATA[
            ...
            Python function that returns an integer = slot number
            ...
       ]]></code>
    </splitter-python>
</window-...>
```

When the `use` element is specified, the ESP server passes the entire input event or a subset of the fields to the specified function.

Specify the `use` element for performance and efficiency. For example, when you have an input event with 100 fields, but you only use them in your Python code, list those fields within the `use` element.

Note: Whenever you have a function named `init` in your Python code, it is called before the model begins to initialize the splitter.

The following project takes trade data and uses slots to send them to one of four Copy windows. The slot is determined by the size of the trade (the `quant` field). The Copy windows are named `small`, `medium`, `large`, and `jumbo` according to the size of the trade. The example also shows the use of a project package that includes an input file.

```
<project name="python_splitter" index="pi_HASH" pubsub="auto"
threads="4">
 <contqueries>
  <contquery name="cq">
   <windows>
    <window-source name="source" insert-only="true">
     <schema>
        <fields>
           <field name="id" type="int64" key="true"/>
           <field name="symbol" type="string" key="false"/>
           <field name="price" type="double" key="false"/>
           <field name="quant" type="int32" key="false"/>
        </fields>
      </schema>
      <connectors>
        <connector class="fs" name="fs">
          <properties>
            <property name="type"><![CDATA[pub]]></property>
            <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/
test_files/trades.csv]]></property>
            <property name="fstype"><![CDATA[csv]]></property>
          </properties>
       </connector>
    </connectors>
      <splitter-python function="getSlot">
        <use>quant</use>
        <code><![CDATA[
def getSlot(data):
    if (data["quant"] < 100):
        return(0)
    elif (data["quant"] < 1000):
        return(1)
    elif (data["quant"] < 10000):
        return(2)
    else:
        return(3)
        ]]></code>
      </splitter-python>
    </window-source>
    <window-copy name="small">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
```

```
    <window-copy name="medium">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="large">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="jumbo">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
   </windows>
   <edges>
    <edge source="source" target="small" slot="0"/>
    <edge source="source" target="medium" slot="1"/>
    <edge source="source" target="large" slot="2"/>
    <edge source="source" target="jumbo" slot="3"/>
   </edges>
  </contquery>
 </contqueries>
</project>
```

The following project is identical to the previous one except that it processes data from a Python connector.

```
<project name="python_splitter2" index="pi_HASH" pubsub="auto"
threads="4">
 <contqueries>
  <contquery name="cq">
   <windows>
    <window-source name="source" insert-only="true">
     <schema>
        <fields>
           <field name="id" type="int64" key="true"/>
           <field name="symbol" type="string" key="false"/>
           <field name="price" type="double" key="false"/>
           <field name="quant" type="int32" key="false"/>
        </fields>
      </schema>
      <connectors>
        <connector class="python" name="Trades">
           <properties>
                <property name="type">pub</property>
                <property name="code"><![CDATA[data = [
    {"id": "991", "symbol": "ibm", "price": "840.68", "quant":
"36099"},
    {"id": "992", "symbol": "goog", "price": "171.69", "quant": "364"},
    {"id": "993", "symbol": "nvda", "price": "460.77", "quant":
"44992"},
    {"id": "994", "symbol": "msft", "price": "929.5", "quant": "5"},
    {"id": "995", "symbol": "amzn", "price": "381.3", "quant":
"18358"},
    {"id": "996", "symbol": "ibm", "price": "496.77", "quant":
"47064"},
    {"id": "997", "symbol": "goog", "price": "480.13", "quant": "700"},
    {"id": "998", "symbol": "nvda", "price": "570.87", "quant":
"33048"},
    {"id": "999", "symbol": "msft", "price": "785.11", "quant":
"3129"},
```

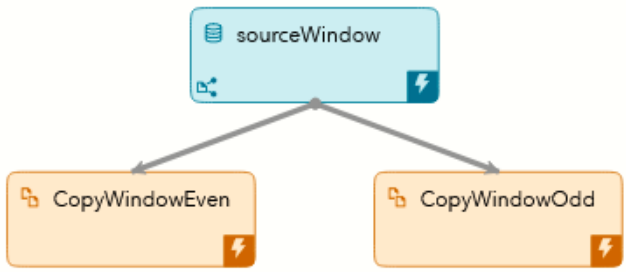
```
    {"id": "1000", "symbol": "amzn", "price": "740.77", "quant":
"48678"},
    {"id": "1001", "symbol": "ibm", "price": "191.07", "quant":
"5790"},
    {"id": "1002", "symbol": "goog", "price": "74.7", "quant":
"36007"},
    {"id": "1003", "symbol": "nvda", "price": "178.28", "quant":
"25722"},
    {"id": "1004", "symbol": "msft", "price": "482.54", "quant":
"18697"},
    {"id": "1005", "symbol": "amzn", "price": "888.2", "quant":
"13810"},
    {"id": "1006", "symbol": "ibm", "price": "715.83", "quant":
"39065"},
    {"id": "1007", "symbol": "goog", "price": "208.4", "quant":
"8644"},
    {"id": "1008", "symbol": "nvda", "price": "607.88", "quant":
"27985"},
    {"id": "1009", "symbol": "msft", "price": "259.78", "quant": "23"},
    {"id": "1010", "symbol": "amzn", "price": "144.64", "quant":
"2383"}
]
def publish():
    return {"events": data, "done": True}
]]></property>
</properties>
       </connector>
    </connectors>
      <splitter-python function="getSlot">
        <use>quant</use>
        <code><![CDATA[
def getSlot(data):
    if (data["quant"] < 100):
        return(0)
    elif (data["quant"] < 1000):
        return(1)
    elif (data["quant"] < 10000):
        return(2)
    else:
        return(3)
        ]]></code>
      </splitter-python>
    </window-source>
    <window-copy name="small">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="medium">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="large">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="jumbo">
      <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
   </windows>
```

```
    <edges>
     <edge source="source" target="small" slot="0"/>
     <edge source="source" target="medium" slot="1"/>
     <edge source="source" target="large" slot="2"/>
     <edge source="source" target="jumbo" slot="3"/>
    </edges>
   </contquery>
  </contqueries>
</project>
```

Using Splitter Functions in Expression Engine Language

The following project consists of one Source window and two Copy windows. The Source window contains a user-defined function as a splitter to determine which of two slots an event should go to. Each slot directs to a different Copy window.

Figure 7.1 *Model That Uses a Splitter*



Here is the Source window:

```
<window-source index="pi_RBTREE" pubsub="true" name="sourceWindow">
          <splitter-expr>
            <expr-initialize>
              <udfs>
                <udf name="udf1" type="int32"><![CDATA[return ID%2]]></
udf>
                <udf name="udf2" type="string"><![CDATA[return
upper(symbol)]]></udf>
                <udf name="udf3" type="double"><![CDATA[return
price]]></udf>
              </udfs>
            </expr-initialize>
            <expression><![CDATA[udf1() and (match_string(udf2(),
"IBM")) and (udf3() > 100.0)]]></expression>
          </splitter-expr>
          <schema>
            <fields>
              <field name="ID" type="int32" key="true"/>
```

```
            <field name="symbol" type="string"/>
            <field name="price" type="double"/>
          </fields>
        </schema>
        <connectors>
          <connector class="fs" name="input_splitter">
            <properties>
              <property name="type"><![CDATA[pub]]></property>
              <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/
test_files/input_slots.csv]]></property>
              <property name="fstype"><![CDATA[csv]]></property>
            </properties>
          </connector>
        </connectors>
</window-source>
```

The Source window uses a file and socket connector to receive input events from a CSV file named **`input.csv`**. Events include an ID, a company's ticker symbol, and a trading price.

```
i,    n,    1,    ibm,    99.3
i,    n,    2,    amzn,    1626.03
u,    n,    2,    amzn,    1629.99
i,    n,    3,    appl,    197
p,    n,    3,    appl,    198
i,    n,    5,    goog,    1094.58
d,    n,    1,    ibm,    95.5
i,    n,    11,    fb,    138.68
u,    n,    11,    fb,    137.5
i,    n,    4,    amzn,    1633.2
i,    n,    9,    ibm,    123.46
u,    n,    9,    ibm,    126.44
p,    n,    3,    appl,    197.2
i,    n,    6,    nflx,    253.9
p,    n,    5,    goog,    1096.8
p,    n,    11,    fb,    139.5
i,    n,    7,    tsla,    180.8
```

The Source window includes a splitter. Three user-defined functions written in *Expression Engine Language* initialize the expression that is used to evaluate which slot to use.

Table 7.1 *Results of User-Defined Functions*

User-Defined Function	Description
`udf1`	Performs a modulo operation on ID. Because ID is an INT32, it returns a whole number. The result of ID mod 2 is 0 when ID is even and 1 when ID is odd. Thus, `udf1` returns a Boolean value **`true`** when ID is odd (remainder 1) and **`false`** when ID is even (remainder 0).
`udf2`	Uses the `match_string` function to check whether the string that is returned contains the substring "IBM."

User-Defined Function	Description
udf3	Compares the numeric output of the function to determine whether it is greater than 100.0.

As each event is streamed, the following expression obtains a result to determine its target slot:

```
udf1() and (match_string(udf2(), "IBM")) and (udf3() > 100.0)
```

Thus, events are routed to slot 1 whenever the share price valuation of IBM is over 100.00.

Edges between the Source and Copy windows specify the slots.

```
<edge source="sourceWindow" target="CopyWindowEven" slot="0"/>
<edge source="sourceWindow" target="CopyWindowOdd" slot="1"/>
```

Here are the Copy windows:

```
<window-copy index="pi_RBTREE" pubsub="true" name="CopyWindowEven">
       <retention type="bycount_sliding"><![CDATA[1000]]></retention>
</window-copy>

<window-copy index="pi_HASH" pubsub="true" name="CopyWindowOdd">
       <retention type="bycount_sliding"><![CDATA[1000]]></retention>
</window-copy>
```

After streaming the incoming events, here are the events that are processed by CopyWindowEven:

```
Insert    7    tsla    180.8
Delete    11    fb    137.5
Update block    11    fb    139.5
Delete    5    goog    1,094.58
Update block    5    goog    1,096.80
Insert    6    nflx    253.9
Delete    3    appl    198
Update block    3    appl    197.2
Insert    4    amzn    1,633.20
Delete    11    fb    138.68
Update block    11    fb    137.5
Insert    11    fb    138.68
Delete    1    ibm    99.3
Insert    5    goog    1,094.58
Delete    3    appl    197
Update block    3    appl    198
Insert    3    appl    197
Delete    2    amzn    1,626.03
Update block    2    amzn    1,629.99
Insert    2    amzn    1,626.03
Insert    1    ibm    99.3
```

Here are the events that are processed by CopyWindowOdd:

```
Delete    1    ibm    123.46
Update block    1    ibm    126.44
Insert    1    ibm    123.46
```

Understanding Retention

Any Source or Copy window can set a retention policy, which governs how it introduces Deletes into the event stream. These Deletes work their way along the data flow, recomputing the model along the way. Internally generated Deletes are flagged with a retention flag, and all further window operations that are based on this Delete are flagged.

Note: Under some circumstances when you map a SAS Micro Analytic Service method to a Calculate window, multiple derived events can be generated. In this case, retention flag propagation does not reliably occur.

For example, consider a Source window with a sliding volume-based retention policy of two. That Source window always contains at most two events. When an Insert arrives causing the Source window to grow to three events, the event with the oldest modification time is removed. A Delete for that event is executed.

Retention Type	Description
time-based	Retention is performed as a function of the age of events. The age of an event is calculated as current time minus the last modification time of the event. Time can be driven by the system time or by a time field that is embedded in the event. A window with time-based retention uses current time set by the arrival of an event. **Note:** The minimum time that you can specify for time-based retention is 1 second.
volume-based	Retention is based on a specified number of records. When the volume increases beyond that specification, the oldest events are removed.

Both time and volume-based retention can occur in one of two variants:

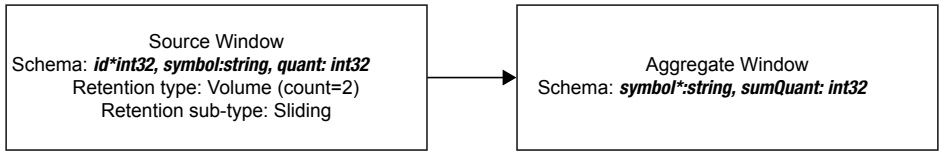
Retention Variant	Description
sliding	Specifies a continuous process of deleting events. Think of the retention window sliding continuously. For a volume-based sliding

Retention Variant	Description
	window, when the specified threshold is hit, one delete is executed for each insert that comes in.
jumping	Specifies a window that completely clears its contents when a specified threshold value is hit. Think of a ten-minute jumping window as one that deletes its entire contents every 10 minutes.

A canonical set of events is a collapsed minimal set of events such that there is at most one event per key. Multiple updates for the same key and insert + multiple updates for the same key are collapsed. A window with retention generates a canonical set of changes (events). Then it appends retention-generated Deletes to the end of the canonical event set. At the end of the process, it forms the final output block.

Windows with retention produce output event blocks of the following form: {<canonical output events>, <canonical retention deletes>}. All other windows produce output blocks of the following form: {<canonical output events>}.

Consider the following model:



The following notation is used to denote events [<opcode>/<flags>: f1, ... ,fn]

- Opcode
  - i — insert
  - d — delete
  - ub — update block — any event marked as ub is always followed by an event marked as d
- Flags
  - n — normal
  - r — retention generated

Suppose that the following events are streamed into the model:

Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 1,ibm,10]	[i/n: 1,ibm,10]	[i/n: ibm,10]
Source In	Source Out — Aggregate In	Aggregate Out

`[i/n: 2,ibm,11]`	`[i/n: 2,ibm,11]`	`[ub/n: ibm,21]` `[d/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 3,sas,100]`	`[i/n: 3,sas,100]`	`[i/n: sas,100]`
	`[d/r: 1,ibm,10]`	`[ub/r: ibm,11]` `[d/r: ibm,21]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 4,ibm,12]`	`[i/n: 4,ibm,12]`	`[ub/r: ibm,12]` `[d/r: ibm,11]`
	`[d/r: 2,ibm,11]`	

When you run in retention-tracking mode, retention and non-retention changes are pulled through the system jointly. When the system processes a user event, the system generates a retention Delete. Both the result of the user event and the result of the retention Delete are pushed through the system. You can decide how to interpret the result. In normal retention mode, these two events can be combined to a single event by rendering its output event set canonical.

Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 1,ibm,10]`	`[i/n: 1,ibm,10]`	`[i/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 2,ibm,11]`	`[i/n: 2,ibm,11]`	`[ub/n: ibm,21]` `[d/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 3,sas,100]`	`[i/n: 3,sas,100]`	`[i/n: sas,100]`
	`[d/r: 1,ibm,10]`	`[ub/r: ibm,11]` `[d/r: ibm,21]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 4,ibm,12]`	`[i/n: 4,ibm,12]`	`[ub: ibm,23]`

	`[d/n: ibm,11]`
`[d/r: 2,ibm,11]`	`[ub/r: ibm,12]` `[d/r: ibm,23]`

Here, the output of the Aggregate window, because of the last input event, is non-canonical. In retention tracking mode, you can have two operations per key when the input events contain a user input for the key and a retention event for the same key.

Note: A window with pulsed mode set always generates a canonical block of output events. For the pulse to function as designed, the window buffers output events until a certain threshold time. The output block is rendered canonical before it is sent.

For examples of retention in practice, consider the following use cases for the four different retention type and variant combinations:

Retention Type and Variant	Use Case Examples
bytime_sliding	■ Real-Time Dashboards ■ Continuously analyzing the most recent events
bytime_jumping	■ Time Period Alarms ■ "Notify us when average pressure hits 2 ATM within the current hour"
bycount_sliding	■ Real-Time Dashboards ■ Continuously analyzing the *n* (100, 200, ...) most recent events
bycount_jumping	■ Analyze or Aggregate events in batches ■ "Aggregate *n* events before loading to minimize impact on the DB"

Understanding Primary and Specialized Indexes

Overview

In order to process events with opcodes, all windows must have a primary index. That index enables the rapid retrieval, modification, or deletion of events in the window.

Some windows have other indexes that serve specialized purposes.

- Source and Copy windows have an additional index to aid in retention
- Join windows have left and right local indexes along with optional secondary indexes. These indexes help avoid locking and maintain data consistency.
- Aggregate windows have an aggregation index to maintain the group structure

Table 7.2 *Specialized Index Types*

Window Type	Retention Index	Aggregation Index	Left Local Index	Right Local Index
Source Window Copy Window	Yes			
Join Window			Yes Optional secondary	Yes Optional secondary
Aggregate Window		Yes		

Fully Stateful Primary Indexes

Any window that uses a fully stateful primary index has a size equal to the cardinality of the unique set of keys. The exception is when a time or size-based retention policy is enforced. Events are absorbed, merged into a window's index, and a canonical version of the change to the index is passed to all output windows.

Table 7.3 Comparison of Fully Stateful Primary Index Types

Primary Index Type	Algorithm	Storage	Advantages	Drawbacks
`pi_RBTREE`	Red-Black Tree	In-memory	Ordered data and memory management provide smooth latencies	Slower than hash
`pi_HASH`	Open Hash	In-memory	Provides faster results than `pi_RBTREE`	Might lead to latency spikes when not properly sized
`pi_HLEVELDB`	B-tree	Local Disk	Big Data ■ Index used for large source or Copy windows and for the left or right local dimension index in a join ■ Can be used when there is no retention or aggregation, or when you need a secondary index	I/O performance **Note:** Do not use`pi_HLEVELDB` where a query, project, or any window name is written in MBCS.
`pi_HLEVELDB_NC`	B-tree	Local Disk	Offers the same advantages for big data storage as `pi_HLEVELDB`, with persistence across restarts	I/O performance

When no retention policy is specified, a window that uses one of the fully stateful indices acts like a database table or materialized view. At any point, it contains the canonical version of the event log. Because common events are reference-counted across windows in a project, you should be careful that all retained events do not exceed physical memory.

Use the Update and Delete opcodes for published events (as is the case with capital market orders that have a life cycle such as create, modify, and close order). However, for events that are Insert-only, you must use window retention policies to keep the event set bound below the amount of available physical memory.

Using pi_HLEVELDB and pi_HLEVELDB_NC Primary Indexes

Overview

The `pi_HLEVELDB` and `pi_HLEVELDB_NC` primary indexes store values on disk and maintain a most-recently used (MRU) in-memory cache. You can use these index types when there is no retention, aggregation, or need for a secondary index.

Note: Do not use `pi_HLEVELDB` or `pi_HLEVELDB_NC` when a query, project, or any window name is written with the variable-width encoding of MBCS.

When you use `pi_HLEVELDB`, you can stream millions of events into a window and consume just a few gigabytes of real memory. However, each time that you load the model (or restart the ESP server), the on-disk index is cleared. For long-lived servers or servers that have the ability to rebuild the index on restart, this might not be optimal usage of system resources. In that case, use the `pi_HLEVELDB_NC` index instead. That index essentially is `pi_HLEVELDB` index that does not clear its on-disk locations upon initialization.

Note: When the ESP server restarts, the on-disk index is reloaded for windows with `pi_HLEVELDB_NC`, but no events are sent to downstream windows.

Using a Large Lookup Table with Persistence across Restarts

The following example shows how to enable an efficient no-regenerate lookup join. Both Source windows are stateless, that is, they have indexes of `pi_EMPTY`.

```
<window-source name="stream_source" index="pi_EMPTY" insert-
only='true'>
               <schema>
                  <fields>
                     <field name='ID' type='int64' key='true'/>
                     <field name='matchID' type='int64'/>
                  </fields>
               </schema>
               <connectors>
                  <connector class='fs' name='pub'>
                     <properties>
                        <property name='type'>pub</property>
```

```
                        <property name='fstype'>csv</property>
                        <property name='fsname'>input/stream.csv</
property>
                        <property name='transactional'>true</property>
                        <property name='blocksize'>1</property>
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                     </properties>
                  </connector>
               </connectors>
            </window-source>

<window-source name='lookup_source' index='pi_EMPTY'>
               <schema>
                  <fields>
                     <field name='ID' type='int64' key='true' />
                     <field name='Lookup' type='string' />
                  </fields>
               </schema>
               <connectors>
                  <connector class='fs' name='pub'>
                     <properties>
                        <property name='type'>pub</property>
                        <property name='fstype'>csv</property>
                        <property name='fsname'>input/500m.csv</
property>
                        <property name='transactional'>true</property>
                        <property name='blocksize'>64</property>
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                     </properties>
                  </connector>
               </connectors>
</window-source>
```

Note:

- The lookup table is stored on disk in a HyperLevel DB
- A single copy of the data exists within the model. Only the in-memory cache is referenced during processing.
- The join is resilient to server restarts, that is, the lookup table does not need to be reloaded if the system bounces.

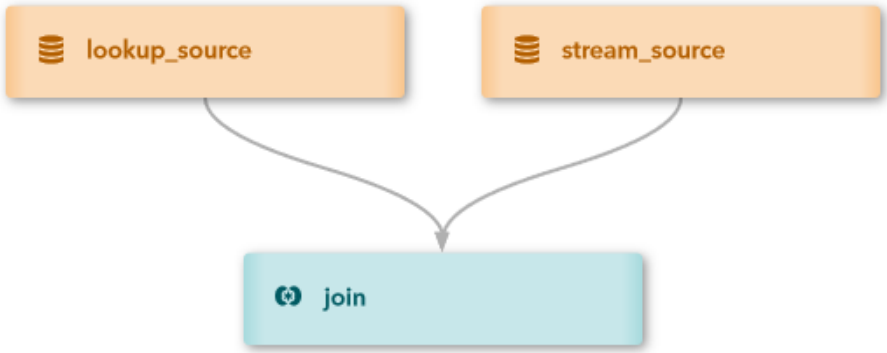
The Join window itself is also stateless. The right-index of the join has an index of `pi_HLEVELDB`. Because this model performs the initial load of the lookup table, you want any old lookup data completely cleared out. Thus, you use `pi_HLEVELDB` and not `pi_HLEVELDB_NC`.

```
<window-join name="join" index='pi_EMPTY'>
               <join type="leftouter" no-regenerates='true' right-
index='pi_HLEVELDB'>
                  <conditions>
                     <fields left="matchID" right="ID" />
                  </conditions>
```

```
                </join>
                <output>
                   <field-selection name="matchID" source="l_matchID" />
                   <field-selection name="lookup" source="r_Lookup" />
                </output>
                <connectors>
                   <connector class='fs' name='sub'>
                      <properties>
                         <property name='type'>sub</property>
                         <property name='fstype'>csv</property>
                         <property name='fsname'>join-out.csv</property>
                         <property name='snapshot'>true</property>
                         <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                      </properties>
                   </connector>
                </connectors>
             </window-join>
```

Edges connect the windows to yield the following:

Figure 7.2 *Continuous Query*



Now suppose that you have 500 million rows of lookup data of the following form:

```
i,n,0, lookup string #0
i,n,1, lookup string #1
i,n,2, lookup string #2
.
.
.
i,n,499999997, lookup string #499999997
i,n,499999998, lookup string #499999998
i,n,499999999, lookup string #499999999
```

Running this model, which loads millions of rows of data, can take several minutes to run. After the lookup data is loaded, you use the following streaming data to test the lookup:

```
i,n,1,1
i,n,2,2
i,n,3,9999999
```

Running those Insert events yields the following results:

```
<event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>1</value>
    <value name='lookup'>lookup string #1</value>
    <value name='matchID'>1</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>2</value>
    <value name='lookup'>lookup string #2</value>
    <value name='matchID'>2</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>3</value>
    <value name='lookup'>lookup string #9999999</value>
    <value name='matchID'>9999999</value>
 </event>
```

Now suppose you stop the project. You run a new one identical to the previous one except that it uses `pi_HLEVELDB_NC` in the join's lookup index.

```
<join type="leftouter" no-regenerates='true' right-
index='pi_HLEVELDB_NC'>
```

You process a few lookup side maintenance events.

```
p,n,14, NEW lookup string #14
u,n,499999999, NEW lookup string #499,999,999
p,n,  4999999, NEW lookup string #4,999,999
d,n, 99999999
```

Then you process a few Insert events to verify that the lookups are performed correctly.

```
i,n,1,1
i,n,2,2
i,n,3,  4999999
i,n,4, 99999999
i,n,5,14
i,n,6,  3333333
i,n,7,499999999
```

The newly joined results are as follows:

```
<event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>1</value>
    <value name='lookup'>lookup string #1</value>
    <value name='matchID'>1</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>2</value>
    <value name='lookup'>lookup string #2</value>
    <value name='matchID'>2</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>3</value>
    <value name='lookup'>NEW lookup string #4</value>
    <value name='matchID'>4999999</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>4</value>
    <value name='matchID'>99999999</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>5</value>
    <value name='lookup'>NEW lookup string #14</value>
    <value name='matchID'>14</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>6</value>
    <value name='lookup'>lookup string #3333333</value>
    <value name='matchID'>3333333</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>7</value>
    <value name='lookup'>NEW lookup string #499</value>
    <value name='matchID'>499999999</value>
 </event>
```

Non-Stateful Primary Index

Any window that uses a primary index type `pi_EMPTY` is non-stateful or stateless. It acts as a pass-through for all incoming events. This index does not store events.

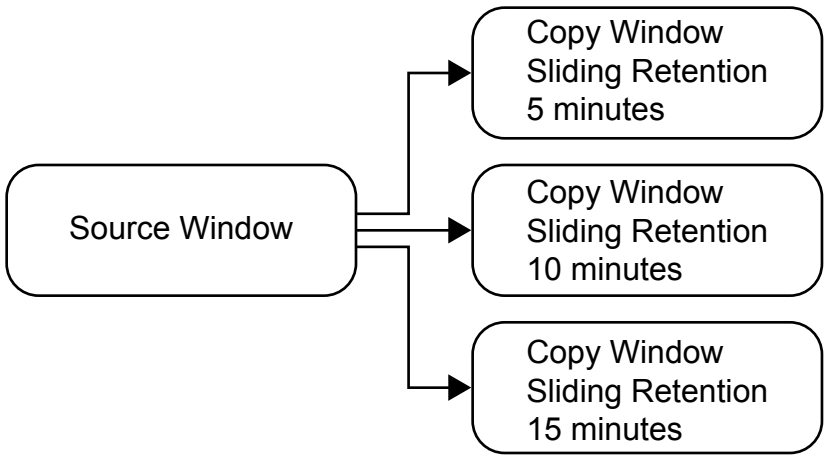
The following restrictions apply to Source windows that use the empty index type.

- No restrictions apply if the Source window is set to "Insert only."
- If the Source window is not Insert-only, then it must be followed by one of the following:
  - a Copy window with a stateful index
  - a Functional window or a Compute window
- When a source, compute, or Functional window in a linear chain with `pi_EMPTY` indexes start a model, one of the following must be true about the linear chain:

- It must end with a functional window that converts its events to Insert only, and have the `produces-only-inserts` property set.
- It must end in a stateful compute, functional, or Copy window that convert Upserts to Inserts. Alternatively, it must have updates that can propagate further through the model automatically through the stateful index.

Using empty indices and retention enables you to specify multiple retention policies from common event streams coming in through a Source window. The source window is used as an absorption point and pass-through to the copy windows, as shown in the following figure.

Figure 7.3 *Copy Windows*

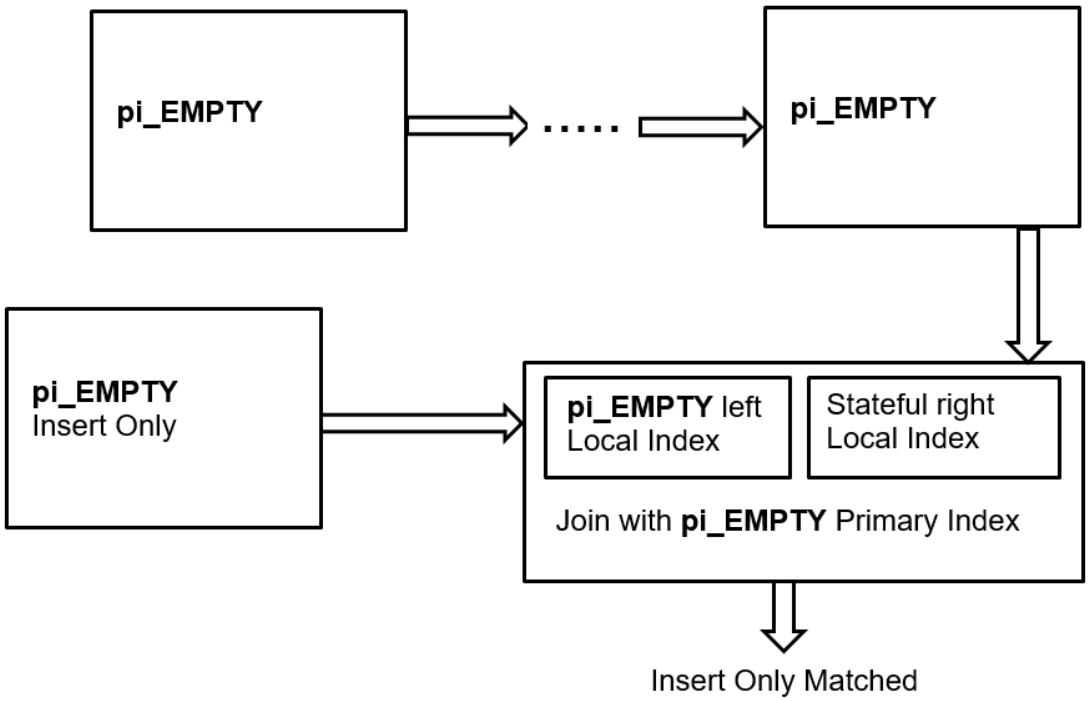


Using a Stateful Local Join Index to Resolve the State

In the past, a streaming join lookup where the lookup table required maintenance through publishing Insert, Update, and Delete events required a large amount of memory. This lookup required a stateful Source window for the lookup side of the join. That stateful Source window fully resolved Insert, Update, and Delete events and fed them to the join. The Insert, Update, and Delete events were applied to the local lookup index in the join. A considerable amount of memory was wasted because of the dual stateful index maintenance (the Source window and the Join window's local lookup index).

Beginning with SAS Event Stream Processing 5.2, the local stateful index for the lookup side of the join can resolve Insert, Update, Upsert, and Delete events. This enables the windows that feed into the lookup side of the join to be pi_EMPTY but still contain Insert, Update, Upsert, and Delete events for lookup maintenance.

Figure 7.4 *Using a Stateful Index for a Left Outer Join*



The following code shows a low memory join:

Note: You must maintain a finite number of events to ensure a bounded memory footprint. The local lookup index for the join maintains all events (minus deleted events) that stream into the lookup side of the join.

Here are the Source windows:

```
<project name='pr_01' pubsub='auto' threads='3' disk-store-path='./'>
   <contqueries>
      <contquery name='cq_01'>
         <windows>
            <window-source name="stream_source" index="pi_EMPTY"
insert-only='true'>
               <schema>
                  <fields>
                     <field name='ID' type='int64' key='true'/>
                     <field name='matchID' type='int64'/>
                  </fields>
               </schema>
               <connectors>
                  <connector class='fs' name='pub'>
                     <properties>
                        <property name='type'>pub</property>
                        <property name='fstype'>csv</property>
                        <property name='fsname'>input/stream.csv</
property>
                        <property name='transactional'>true</property>
                        <property name='blocksize'>1</property>
```

```
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                     </properties>
                  </connector>
                  <connector class='fs' name='pub1'>
                     <properties>
                        <property name='type'>pub</property>
                        <property name='fstype'>csv</property>
                        <property name='fsname'>input/stream-1.csv</
property>
                        <property name='transactional'>true</property>
                        <property name='blocksize'>1</property>
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                     </properties>
                  </connector>
               </connectors>
            </window-source>

            <window-source name='lookup_source' index='pi_EMPTY'>
               <schema>
                  <fields>
                     <field name='ID' type='int64' key='true' />
                     <field name='Lookup' type='string' />
                  </fields>
               </schema>
               <connectors>
                  <connector class='fs' name='pub'>
                     <properties>
                        <property name='type'>pub</property>
                        <property name='fstype'>csv</property>
                        <property name='fsname'>input/10.csv</property>
                        <property name='transactional'>true</property>
                        <property name='blocksize'>64</property>
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                     </properties>
                  </connector>
                  <connector class='fs' name='pub1'>
                     <properties>
                        <property name='type'>pub</property>
                        <property name='fstype'>csv</property>
                        <property name='fsname'>input/10-update.csv</
property>
                        <property name='transactional'>true</property>
                        <property name='blocksize'>64</property>
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                     </properties>
                  </connector>
               </connectors>
            </window-source>
```

Here is the Join window:

```
            <window-join name="join" index='pi_EMPTY'>
```

```
                <join type="leftouter" no-regenerates='true' right-
index='pi_HASH'>
                   <conditions>
                      <fields left="matchID" right="ID" />
                   </conditions>
                </join>
                <output>
                   <field-selection name="matchID"
source="l_matchID" />
                   <field-selection name="lookup" source="r_Lookup" />
                </output>
                <connectors>
                   <connector class='fs' name='sub'>
                      <properties>
                         <property name='type'>sub</property>
                         <property name='fstype'>csv</property>
                         <property name='fsname'>join-out.csv</
property>
                         <property name='snapshot'>true</property>
                        <property name="dateformat">%Y-%m-%d %H:%M:%S</
property>
                      </properties>
                   </connector>
                </connectors>
             </window-join>
          </windows>
          <edges>
             <edge source="lookup_source" target="join" role="right" />
             <edge source="stream_source" target="join" role="left"  />
          </edges>
       </contquery>
    </contqueries>
```

Here are the connector groups and edges:

```
<project-connectors>
      <connector-groups>
         <connector-group name="group1">
            <connector-entry connector='cq_01/join/sub'
state='running'/>
            <connector-entry connector='cq_01/lookup_source/pub'
state='finished'/>
         </connector-group>
         <connector-group name="group2">
            <connector-entry connector='cq_01/stream_source/pub'
state='finished'/>
         </connector-group>
         <connector-group name="group3">
            <connector-entry connector='cq_01/lookup_source/pub1'
state='finished'/>
         </connector-group>
         <connector-group name="group4">
            <connector-entry connector='cq_01/stream_source/pub1'
state='finished'/>
         </connector-group>
      </connector-groups>
      <edges>
```

```
            <edge source='group1' target='group2'/>
            <edge source='group2' target='group3'/>
            <edge source='group3' target='group4'/>
        </edges>
    </project-connectors>
</project>
```

Four input data files are orchestrated in the following way:

1. The first file contains data that is fed to the lookup side of the join.

```
i,n,0, lookup string #0
i,n,1, lookup string #1
i,n,2, lookup string #2
i,n,3, lookup string #3
i,n,4, lookup string #4
i,n,5, lookup string #5
i,n,6, lookup string #6
i,n,7, lookup string #7
i,n,8, lookup string #8
i,n,9, lookup string #9
```

2. The second file contains streaming data that is fed to the streaming side of the join, and join output is produced.

```
i,n,1,1
i,n,2,2
i,n,3,9
```

   Here is the initial output of the join:

```
I,N, 1,1,lookup string #1
I,N, 2,2,lookup string #2
I,N, 3,9,lookup string #9
```

3. The third file contains a second set of data that is fed to the lookup side of the join. Join maintenance occurs; Inserts, Updates, Upserts, and Deletes are performed on the lookup table.

```
u,n,0, NEW lookup string #0
d,n,5,
p,n,9, NEW lookup string #9
d,n,17
```

4. The fourth file contains a second set of streaming data that is fed to the streaming side of the join. The lookups reflect the previously executed join maintenance.

```
i,n,1,1
i,n,2,2
i,n,3,9
i,n,4,5
i,n,5,0
```

   Here is the output of the join after lookup table maintenance has been applied:

```
I,N, 1,1,lookup string #1
I,N, 2,2,lookup string #2
I,N, 3,9,NEW lookup string #9
I,N, 4,5,
I,N, 5,0,NEW lookup string #0
```

Restrictions on a Window's Primary Index and Input Windows

When you violate the following restrictions on a window's primary index or on its input windows, the ESP server returns a fatal error that is noted in the log. As a result, your model fails to start. This flags improper models before they process data and cause run-time problems.

Note: Sometimes when a window has a stateful index and receives only inserts, the ESP server returns a fatal error that memory will grow indefinitely. In those cases, your model fails to start.

Table 7.4 *Restrictions on a Window's Primary Index and Input Windows*

Window	Index Restriction	Input Window Restriction	Opcodes Output
Aggregate	Use only stateful indexes	Input window cannot have `pi_EMPTY` index and cannot produce non-Inserts	All
Calculate	None	Input window cannot have `pi_EMPTY` index and produce non-Inserts	When `algorithm='MAS'`, set `produces-only-inserts='true'`. Otherwise, produce only Inserts when all input windows produce only Inserts
Compute	None	None	Produces only Inserts when the input window produces only Inserts
Copy	Use only stateful indexes; this requires that	None	All

Window	Index Restriction	Input Window Restriction	Opcodes Output
	retention is set. See Note.		
Counter	None	None	Produces only Inserts when the index is `pi_EMPTY`
Filter	None	Input window cannot be `pi_EMPTY` and produce non-Inserts	Produces only Inserts when input window produces only Inserts
Functional	None	None	Produces only Inserts when all input windows produce only Inserts
Geofence	None	None	When the event position input window always produces Inserts, only Inserts are produced
Join	None	None	When the streaming side of the join produces only Inserts and the join is `no-regenerates='true'`, only Inserts are produced
Lua	None	None	Produces only Inserts when the input window produces only Inserts. You can override this default behavior by setting `produces-only-inserts='false'`, which enables the Lua window to generate non-Insert opcodes even when the input window

Window	Index Restriction	Input Window Restriction	Opcodes Output
			produces only Inserts.
Model Reader	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Model Supervisor	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Object Tracking	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Pattern	Use `pi_EMPTY` exclusively	Should receive only Inserts, logs warning on non-Inserts	Always produces Inserts
Procedural	None	None	Set `produces-only-inserts='true'` when appropriate, which depends on the procedural code that executes within the window.
Python	None	None	Produces only Inserts when the input window produces only Inserts. You can override this default behavior by setting `produces-only-inserts='false'`, which enables the Python window to generate non-Insert opcodes even when the input window produces only Inserts.
Remove State	Use `pi_EMPTY` exclusively	None	Always produces Inserts

Window	Index Restriction	Input Window Restriction	Opcodes Output
Score	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Source	None	There are no input windows to a Source window.	Produces only Inserts when set to Insert-only
StateDB Reader	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Produces Inserts if `generate-deletes = 'false'`. Produces inserts and deletes if `generate-deletes = 'true'`.
StateDB Writer	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Text Category	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Text Context	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Text Sentiment	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Text Topic	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Train	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Transpose	Use `pi_EMPTY` exclusively	All input windows must produce only Inserts	Always produces Inserts
Union	Depends on input windows and strict setting	Depends on index type	If any input window produces unresolved Updates

Window	Index Restriction	Input Window Restriction	Opcodes Output
			or Deletes, then the index must be stateful. If all input windows always produce Inserts and the union is strict, then only Inserts are produced; the index can be `pi_EMPTY`.

Note: A Copy window that receives only Inserts and that uses splitter expressions can use a `pi_EMPTY` index.

Understanding Design Patterns

Overview to Design Patterns

A design pattern is a reusable solution to a common problem within a specific context of software design. The combinations of windows that you use in your design pattern should enable fast and efficient event stream processing.

Event stream processing models can be stateless, stateful, or mixed. The type of model that you choose affects how you design it. One challenge when you design a mixed model is to identify sections that must be stateful and those that can be stateless, and then connecting them properly.

A stateless model is one where the indexes on all windows have the type **`pi_EMPTY`**. Events are not retained in any window, and are essentially transformed and passed through. Stateless models exhibit fast performance and use very little memory. They are well-suited to tasks where the inputs are inserts and when simple filtering, computation, text context analysis, or pattern matching are the only operations you require.

A stateful model is one that uses windows with index types that store data, usually **`pi_RBTREE`** or **`pi_HASH`**. These models can fully process events with Insert, Update, or Delete opcodes. A stateful model facilitates complex relational operations such as joins and aggregations. Because events are retained in indexes, whenever all events are Inserts only, windows grow unbounded in memory. Thus, stateful models must process a mix of Inserts, Updates, and Deletes in order to remain bounded in memory.

The mix of opcodes can occur in one of two ways:

- The data source and input events have bounded key cardinality. That is, there are a fixed number of unique keys (such as customer IDs) in the input stream. You can make many updates to these keys provided that the key cardinality is finite.
- A retention policy is enforced for the data flowing in, where the amount of data is limited by time or event count. The data is then automatically deleted from the system by the generation of internal retention delete events.

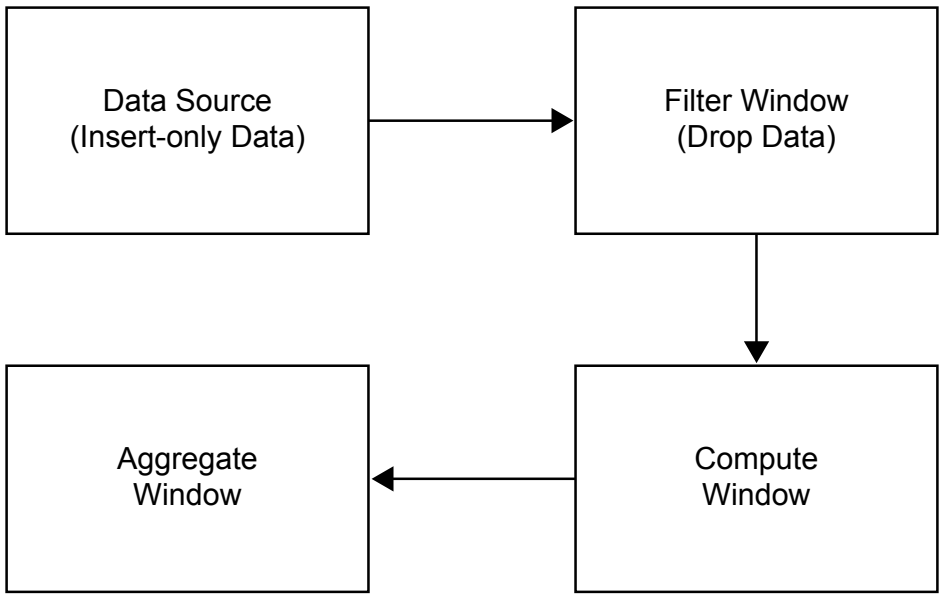
A mixed model has stateless and stateful parts. Often it is possible to separate the parts into a stateless front end and a stateful back end.

Design Pattern That Links a Stateless Model with a Stateful Model

To control memory growth in a mixed model, link the stateless and stateful parts with copy windows that enforce retention policies. Use this design pattern when you have insert-only data that can be pre-processed in a stateless way. Pre-process the data before you flow it into a section of the model that requires stateful processing (using joins, aggregations, or both).

For example, consider the following model:

Figure 7.5 *Event Stream Processing Model with Insert-Only Data*



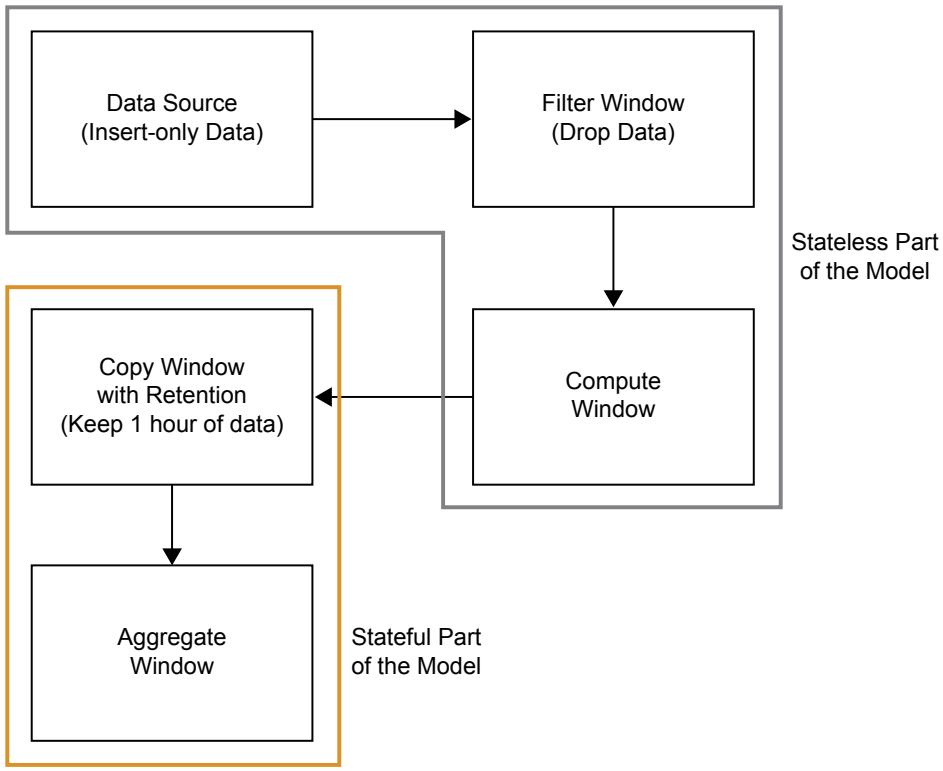
Here the data source is purely through Inserts. Therefore, the model can be made stateless by using an index type of **`pi_EMPTY`**. The filter receives inserts from the source, and drops some of them based on the filter criteria, so it produces a set of inserts as output. Thus, the filter can be made stateless also by using an index type of **`pi_EMPTY`**.

The Compute window transforms the incoming inserts by selecting some of the output fields of the input events. The same window computes other fields based on values of the input event. It generates only inserts, so it can be stateless.

After the Compute window, there is an Aggregate window. This window type needs to retain events. Aggregate windows group data and compress groups into single events. If an Aggregate window is fed a stream of Inserts, it would grow in an unbounded way.

To control this growth, you can connect the two sections of the model with a Copy window with a retention policy.

Figure 7.6 *Modified Event Stream Processing Model with Stateless and Stateful Parts*



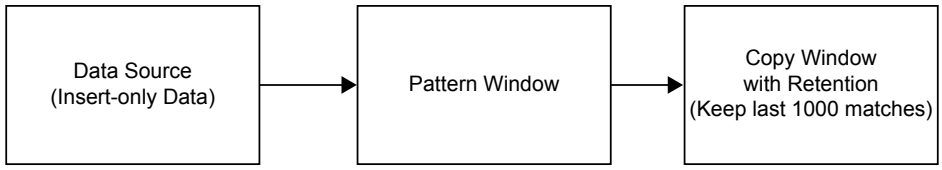
The stateful part of the model is accurately computed, based on the rolling window of input data. This model is bounded in memory.

Controlling Pattern Window Matches

Pattern matches that are generated by Pattern windows are Inserts. Suppose you have a source window feeding a Pattern window. Because a Pattern window generate Inserts only, you must make it stateless by specifying an index type of `pi_EMPTY`. This prevents the Pattern window from growing infinitely. Normally, you want to keep some of the more recent pattern matches around. Because you do not know how frequent the pattern generates matches, follow the pattern window with a count-based Copy window.

Suppose you specify to retain the last 1000 pattern matches in the Copy window.

Figure 7.7 *Event Stream Processing Model with Copy with Retention*



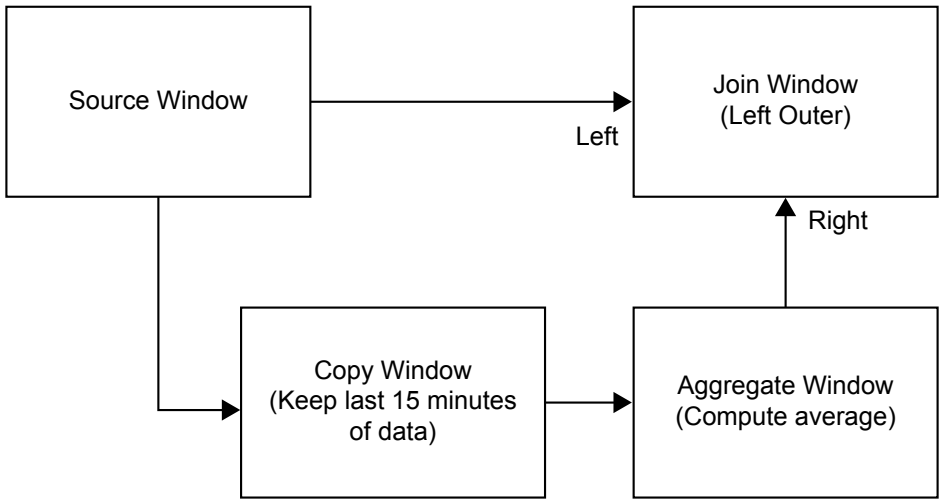
In cases like these, it is more likely that the Copy window is queried from the outside using adapters, or publish/subcribe clients. The Copy window might also feed other sections of the model.

Augmenting Incoming Events with Rolling Statistics

Suppose you have an insert stream of events, and one or more values are associated with the events. You want to augment each input event with some rolling statistics and then produce the augmented events. Solving this problem requires using advanced features of the modeling environment.

For example, suppose you have a stream or stock trades coming in and you want to augment them with the average stock price in the past. You build the following model.

Figure 7.8 *Event Stream Processing Model Using Advanced Features*



To control the aggregate window:

- Put retention before it (the Copy window).
- Group it by symbol (which is bounded), and use the additive aggregation function average (`ESP_aAve`), which does not need to store each event for its computation.

The Join window can be problematic. Ordinarily you think of a Join window as stateful. A join retains data for its fact window or dimension window and performs matches. In this case, you want a special, but frequently occurring behavior. When input comes in, pause it at the join until the aggregate corresponding to the input is processed. Then link the two together, and pass the augmented insert out.

To process input in this way:

1. Make the join a left outer join, with the source feed the left window, and the aggregate feeds the right window.
2. Set Tagged Token data flow model on for the projects. This turns on a special feature that causes a fork of a single event to wait for both sides of the fork to rejoin before generating an output event.
3. Set the index type of the join to **`pi_EMPTY`**, making the join stateless. A stateless left outer join does not create a local copy of the left driving window (FACT window). It does not keep any stored results of the join. However, there is always a reference-counted copy the lookup window. In the case of a left outer join, this is the right window. The lookup window is controlled by retention in this case, so it is bounded.
4. Ordinarily, a join, when the dimension window is changed, tries to find all matching events in the fact window and then issue updates for those joined matches. You do not want this behavior, because you are matching events in lock step. Further, it is simply not possible because you do not store the fact data. To prevent this regeneration on each dimension window change, set the **`no-regenerates`** option on the Join window.

In this way you create a fast, lightweight join. This join stores only the lookup side, and produces a stream of inserts on each inserted fact event. A stateless join is possible for left and right outer joins.

The following XML code implements this model.

```
<project name='trades_proj' pubsub='auto'
        use-tagged-token='true' threads='4'>
            <contqueries>
                <contquery name='trades_cq'>

                    <windows>
                        <window-source name='Trades'
                                       insert-only='true'
                                       index='pi_EMPTY'>
                            <schema>
                                <fields>
                                    <field name='tradeID'
type='string' key='true'/>
                                    <field name='security'
type='string'/>
                                    <field name='quantity'
type='int32'/>
                                    <field name='price' type='double'/>
                                    <field name='traderID'
type='int64'/>
                                    <field name='time' type='stamp'/>
                                </fields>
```

```
                                </schema>
                            </window-source>

                            <window-copy name='TotalIn'>
                                <retention
type='bytime_sliding'>15{minutes}</retention>
                            </window-copy>

                            <window-aggregate name='RunTotal'>
                                <schema>
                                    <fields>
                                        <field name='tradeID'
type='string'/>
                                        <field name='security'
type='string' key='true'/>
                                        <field name='quantityTotal'
type='double'/>
                                    </fields>
                                </schema>
                                <output>
                                    <field-expr>ESP_aLast(tradeID)</field-
expr>
                                    <field-expr>ESP_aSum(quantity)</field-
expr>
                                </output>
                            </window-aggregate>

                            <window-join name='JoinTotal'
                                         index='pi_RBTREE'>
                                <join type="leftouter"
                                      no-regenerates='true'>
                                    <conditions>
                                      <fields left='tradeID'
right='tradeID'/>
                                      <fields left='security'
right='security'/>
                                   </conditions>
                                </join>
                                <output>
                                    <field-selection name='quantity'
                                      source='l_quantity'/>
                                    <field-selection name='price'
                                      source='l_price'/>
                                    <field-selection name='traderID'
                                      source='l_traderID'/>
                                    <field-selection name='time'
                                      source='l_time'/>
                                    <field-selection name='quantityTotal'
                                      source='r_quantityTotal'/>
                                </output>
                            </window-join>
                        </windows>

                        <edges>
                            <edge source='Trades'
                                  target='JoinTotal'/>
```

```
                    <edge source='Trades'
                          target='TotalIn'/>
                    <edge source='TotalIn'
                          target='RunTotal'/>
                    <edge source='RunTotal'
                          target='JoinTotal'/>
                </edges>

            </contquery>
        </contqueries>
    </project>
```

Advanced Window Operations

Writing Aggregate Functions to Embed in Applications

Overview

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

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

After you write an aggregate function, embed it in C++ code in order to use it in your event stream processing application. Suppose your function is named **`My_Aggregation_Function`**. At the bottom of **`functions.cpp`**, create a wrapper function for your aggregation function.

```
// the uMyFunction wrapper:
     // every aggregation function must be wrapped like this.
    //
    int dfESPaggrfunc_uMyFunctionWrapper(dfESPexpEngine::exp_engine_t
*e,

dfESPexpEngine::exp_sym_value_t *returnval,
                                          int parmcount,

dfESPexpEngine::exp_sym_value_t **parms)
```

```
                                                {
        return dfESPaggrfunc_Wrapper((void *)my_aggrergation_function,
e,
                                                   returnval, parmcount,
parms);
                                                }
```

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

```
// Get all user-defined aggregation functions during initialization
    //
    void add_user_aggrFunctions() {
        dfESPengine *e = dfESPengine::getEngine();

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

        // push back as many user defined functions as you like:
        //   the paramters are: <callable name>, <function pointer>,
        //   <arg type vect>, <additive flag>,  <additive flag for
insert only>,
        //   <description>
        uFuncts.push_back(new aggr_function_t("USER_uSum_nadd",
           (void *)dfESPaggrfunc_uSumNAddWrapper, "a", false,
            false, "summarization"));
        uFuncts.push_back(new aggr_function_t("USER_uSum_add",
           (void *)dfESPaggrfunc_uSumAddWrapper, "a", true,
            true, "summarization"));
```

Adjust the number of arguments, additive flags, and the description field accordingly.

Writing Additive Aggregate Functions

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

An additive aggregate function can be complex for two reasons:

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

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

The following code performs these basic steps:

1 Look at the input and output types and verify compatibility.

2 Initialize a return variable of the specified output type.

3 Determine whether the function has been called before. That is, is there a previous state value?

- If so, retrieve it for use.
- If not, create a new group with an arriving insert so that you can set the state to the incoming value.

4 Switch on the opcode and adjust the state value.

5 Check for computational errors and return the error value or the state value as the result.

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

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

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

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

        bool te=false;
        int64_t fID = exp_param_getI64(fid, te);
        if (te) {
          cerr << "could not obtain the integer field offset (field
ID)" << endl;
          rdv = new dfESPdatavar(aType); rdv->null();
          return rdv;
        }

        dfESPdatavar::dfESPdatatype iType = iSchema->getTypeIO(fID);

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

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

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

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

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

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

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

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

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

        // If the state has never been set, we set it and return.
        //
```

```
        if (state.empty()) {
            dv_assign(rdv, &iNdv);
            //  result = input
            state.push_back(new dfESPdatavar(rdv));
            //  make a copy and push as state
            return rdv;
        }

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

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

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

        return rdv;
    }
```

You can use the `Sum()` aggregate function to iterate over the group and compute a new sum when a new group changes. Faster results are obtained when you maintain the `Sum()` in a `dfESPdatavar` in the `dfESPgroupstate` object and increment or

decrement the object by the incoming value, provided the new event is an Insert, Update, or Delete. The function then adjusts this field state so that it is up-to-date and can be used again when another change to the group occurs.

Writing Non-Additive Aggregate Functions

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

The following code performs these basic steps:

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

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

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

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

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


        bool te=false;
        int64_t fID = exp_param_getI64(fid, te);
        if (te) {
          cerr << "could not obtain the integer field offset (field
ID)" << endl;
          rdv = new dfESPdatavar(aType); rdv->null();
          return rdv;
```

```
        }

        dfESPdatavar::dfESPdatatype iType = iSchema->getTypeIO(fID);
        dvn_error_t   retCode = dvn_noError;
        // return code for using the datavar numerics package.

        // If the input fields or the output field is non-numeric,
        // flag an error.
        //
        if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
        cerr << "summation must work on numeric input, produce numeric
output."
             << endl;
        return NULL;
        }

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

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

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


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

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

        if (retCode != dvn_noError ) {       // if any of our
arithmitic fails.
```

```
            rdv->null();                          // return a null value.
            cerr << "uSum() got an arithmetic error in summing up values"
<< endl;
          }

          return rdv;
```

Implementing Periodic (or Pulsed) Window Output

In most cases, the SAS Event Stream Processing API is fully event driven. That is, windows continuously produce output as soon as they transform input. But there might be times when you want a window to hold data and then write a canonical batch of updates. In this case, operations to common key values are collapsed into a single operation.

Here are two cases where batched output might be useful:

- Visualization clients might want to get updates once a second because they cannot visualize changes any faster than this. When the event data is pulsed, the clients take advantage of the reduction of event data to visualize through the collapse around common key values.
- A window that follows the pulsed window is interested in comparing the deltas between periodic snapshots from that window.

Use the following call to add output pulsing to a window:

```
dfESPwindow::setPulseInterval(size_t us);
```

Note: Periodicity is specified in microseconds. However, given the clock resolution of most non-real-time operating systems, the minimum value that you should specify for a pulse period is 100 milliseconds. In your XML code, use the pulse-interval attribute of the window. The value defaults to milliseconds.

Marking Events as Partial-Update on Publish

Overview

In most cases, events are published with all fields available. Some of the field values might be null. Events with Delete opcodes require only the key fields to be non-null.

There are times when only the key fields and the fields being updated are desired or available for event updates. This is typical for financial feeds. For example, a broker

might want to update the price or quantity of an outstanding order. You can update selected fields by marking the event as partial-update (rather than normal).

When you mark events as partial-update, you provide values only for the key fields and for fields that are being updated. In this case, the fields that are not updated are marked as data type `dfESPdatavar::ESP_LOOKUP`. This marking tells SAS Event Stream Processing to match key fields of an event retained in the system with the current event and not to update the current event's fields.

In order for a published event to be tagged as a partial-update, the event must contain all non-null key fields that match an existing event in the Source window. Partial updates are applied to Source windows only.

When using transactional event blocks that include partial events, be careful that all partial updates are for key fields that are already in the Source window. You cannot include the insert of the key values with an update to the key values in a single event block with transactional properties. This attempt fails and is logged because transactional event blocks are treated atomically. All operations in that block are checked against an existing window state before the transactional block is applied as a whole.

Publishing Partial Events into a Source Window

Consider these three points when you publish partial events into a Source window.

- In order to construct the partial event, you must represent all the fields in the event. Specify either the field type and value or a placeholder field that indicates that the field value and type are missing. In this way, the existing field value for this key field combination remains for the updated event. These field values and types can be provided as `datavars` to build the event. Alternatively, they can be provided as a comma-separated value (CSV) string.

  If you use CSV strings, then use '^U' (such as, control-U, decimal value 21) to specify that the field is a placeholder field and should not be updated. On the other hand, if you use `datavars` to represent individual fields, then those fully specified fields should be valid. Enter them as `datavars` with values (non-null or null). Specify the placeholder fields as empty `datavars` of type `dfESPdatavar::ESP_LOOKUP`.

- No matter what form you use to represent the field values and types, the representation should be included in a call for the partial update to be published. In addition to the fields, use a flag to indicate whether the record is a normal or partial update. If you specify partial update, then the event must be an Update or an Upsert that is resolved to an Update. Using partial-update fields makes sense only in the context of updating an existing or retained Source window event. This is why the opcode for the event must resolve to Update. If it does not resolve to Update, an event merge error is generated.

  If you use an event constructor to generate this binary event from a CSV string, then the beginning of that CSV string contains "`u,p`" to show that this is a partial-update. If instead, you use `event->buildEvent()` to create this partial update event, then you need to specify the event flag parameter as

dfESPeventcodes::ef_PARTIALUPDATE and the event opcode parameter as dfESPeventcodes::eo_UPDATE.

- One or more events are pushed onto a vector and then that vector is used to create the event block. The event block is then published into a Source window. For performance reasons, each event block usually contains more than a single event. When you create the event block, you must specify the type of event block as transactional or atomic using dfESPeventblock::ebt_TRANS or as normal using dfESPeventblock::ebt_NORMAL.

  Do not use transactional blocks with partial updates. Such usage treats all events in the event block as atomic. If the original Insert for the event is in the same event block as a partial Update, then it fails. The events in the event block are resolved against the window index before the event block is applied atomically. Use normal event blocks when you perform partial Updates.

Examples

Here are some sample code fragments for the variations on the three points described in the previous section.

Create a partial Update datavar and push it onto the datavar vector.

```
// Create an empty partial-update datavar.
dfESPdatavar* dvp  = new dfESPdatavar(dfESPdatavar::ESP_LOOKUP);
// Push partial-update datavar onto the vector in the appropriate
// location.
// Other partial-update datavars might also be allocated and pushed to
the
// vector of datavars as required.
dvVECT.push_back(dvp); // this would be done for each field in the
update
     event
```

Create a partial Update using partial-update and normal datavars pushed onto that vector.

```
// Using the datavar vector partially defined above and schema,
// create event.
dfESPeventPtr eventPtr = new dfESPevent();
eventPtr->buildEvent(schemaPtr, dvVECT, dfESPeventcodes::eo_UPDATE,
dfESPeventcodes::ef_PARTIALUPDATE);
```

Define a partial update event using CSV fields where '^U' values represent partial-update fields. Here you are explicitly showing '^U'. However, in actual text, you might see the character representation of Ctrl-U because individual editors show control characters in different ways.

Here, the event is an Update (due to 'u'), which is partial-update (due to 'p'), key value is 44001, "ibm" is the instrument that did not change. The instrument is included in the field. The price is 100.23, which might have changed, and 3000 is the quantity, which might have changed, so the last three of the fields are not updated.

```
p = new dfESPevent(schema_01,
```

```
(char *)"u,p,44001,ibm,100.23,3000,^U,^U,^U");
```

Implementing Persist and Restore Operations

SAS Event Stream Processing enables you to do the following:

- persist a complete model state to a file system
- restore a model from a persist directory that had been created by a previous persist operation
- persist and restore an entire engine
- persist and restore a project

To create a persist object for a model, provide a pathname to the class constructor: `dfESPpersist(char *baseDir);` The *baseDir* parameter can point to any valid directory, including disks shared among multiple running event stream processors.

After providing a pathname, call either of these two public methods:

```
bool persist();
bool restore(bool dumpOnly=false);
// dumpOnly = true means do not restore, just walk and print info
```

The `persist()` method can be called at any time. Be aware that it is expensive. Event block injection for all projects is suspended, all projects are quiesced, persist data is gathered and written to disk, and all projects are restored to normal running state.

The `restore()` method should be invoked only before any projects have been started. If the persist directory contains no persist data, the `restore()` call does nothing.

The persist operation is also supported by the C, Java, and Python publish/subscribe APIs. These API functions require a *host:port* parameter to indicate the target engine.

The C publish/subscribe API method is as follows: `int C_dfESPpubsubPersistModel(char *hostportURL, const char *persistPath)`

The Java publish/subscribe API method is as follows: `boolean persistModel(String hostportURL, String persistPath)`

One application of the persist and restore feature is saving state across event stream processor system maintenance. In this case, the model includes a call to the `restore()` function described previously before starting any projects. To perform maintenance at a later time on the running engine:

1. Pause all publish clients in a coordinated fashion.
2. Make one client execute the publish/subscribe persist API call described previously.
3. Bring the system down, perform maintenance, and bring the system back up.

4 Restart the event stream processor model, which executes the `restore()` function and restores all windows to the states that were persisted in step 2.

5 Resume any publishing clients that were paused in step 1.

To persist an entire engine, use the following functions:

```
bool dfESPengine::persist(const char * path);

void dfESPengine::set_restorePath(const char *path);
```

The path that you specify for `persist` can be the same as the path that you specify for `set_restorePath`.

To persist a project, use the following functions:

```
bool dfESPproject::persist(const char *path)

bool dfESPproject::restore(const char *path);
```

Start an engine and publish data into it before you persist it. It can be active and receiving data when you persist it.

To persist an engine, `call dfESPengine::persist(path);`. The system does the following:

1 pauses all incoming messages (suspends publish/subscribe)

2 finish processing any queued data

3 after all queued data has been processed, persist the engine state to the specified directory, creating the directory if required

4 after the engine state is persisted, resume publish/subscribe and enable new data to flow into the engine

To restore an engine, initialize it and call `dfESPengine::set_restorePath(path);`. After the call to `dfESPengine::startProjects()` is made, the entire engine state is restored.

To persist a project call `dfESPproject::persist(path);`. The call turns off publish/subscribe, quiesces the system, persists the project, and then re-enables publish/subscribe. The path specified for restore is usually the same as that for persist.

To restore the project, call `dfESPproject::restore(path);` before the project is started. Then call `dfESPengine::startProject(project);`

Note: When you persist a model that trains online projects, note that the Train window immediately starts training a new model upon model restore. No user intervention is required.

Note: When you persist a model that performs offline training (whereby the model is published to the ESP server through a Model Reader window) and then restore

the model, you must republish the model to the Model Reader window to enable the Score window to score new events.

Gathering and Saving Latency Measurements

The `dfESPlatencyController` class supports gathering and saving latency measurements on an event stream processing model. Latencies are calculated by storing 64-bit microsecond granularity timestamps inside events that flow through windows enabled for latency measurements.

In addition, latency statistics are calculated over fixed-size aggregations of latency measurements. These measurements include average, minimum, maximum, and standard deviation. The aggregation size is a configurable parameter. You can use an instance of the latency controller to measure latencies between any Source window and some downstream window that an injected event flows through.

The latency controller enables you to specify an input file of event blocks and the rate at which those events are injected into the Source window. It buffers the complete input file in memory before injecting to ensure that disk reads do not skew the requested inject rate.

Specify an output text file that contains the measurement data. Each line of this text file contains statistics that pertain to latencies gathered over a bucket of events. The number of events in the bucket is the configured aggregation size. Lines contain statistics for the next bucket of events to flow through the model, and so on.

Each line of the output text file consists of three tab-separated columns. From left to right, these columns contain the following:

- the maximum latency in the bucket
- the minimum latency in the bucket
- the average latency in the bucket

You can configure the aggregation size to any value less than the total number of events. A workable value is something large enough to get meaningful averages, yet small enough to get several samples at different times during the run.

If publish/subscribe clients are involved, you can also modify publisher/subscriber code or use the file/socket adapter to include network latencies as well.

To measure latencies inside the model only:

1. Include "`int/dfESPlatencyController.h`" in your model, and add an instance of the `dfESPlatencyController` object to your `main()`.
2. Call the following methods on your `dfESPlatencyController` object to configure it:

Method	Description
`void set_playbackRate(int32_t r)`	Sets the requested inject rate.
`void set_bucketSize(int32_t bs)`	Sets the bucketSize parameter previously described.
`void set_maxEvents(int32_t me)`	Sets the maximum number of events to inject.
`void set_oFile(char *ofile)`	Sets the name of the output file containing latency statistics.
`void set_iFile(char *ifile)`	Sets the name of the input file containing binary event block data.
`void set_stampBlkSize(int64_t stampsize)`	Specifies the block size (in number of events) of memory to allocate for storing timestamps when in latency mode. Additional blocks are allocated as required.
`void set_skipSize(int32_t ss)`	Specifies the number of beginning and ending aggregation blocks to ignore in latency calculations.

3 Add a subscriber callback to the window where you would like the events to be timestamped with an ending timestamp. Inside the callback add a call to this method on your `dfESPlatencyController` object: `void record_output_events(dfESPeventblock *ob)`. This adds the ending timestamp to all events in the event block.

4 After starting projects, call these methods on your `dfESPlatencyController` object:

Method	Description
`void set_injectPoint(dfESPwindow_source *s)`	Sets the Source window in which you want events time stamped

Method	Description
	with a beginning timestamp.
`void read_and_buffer()`	Reads the input event blocks from the configured input file and buffers them.
`void playback_at_rate()`	Time stamps input events and injects them into the model at the configured rate, up to the configured number of events.

5 Quiesce the model and call this method on your `dfESPlatencyController`object: `void generate_stats()` and pass 4 and 0 for the `metaHigh` and `metaLow` parameters respectively. This gathers the start and end timestamps from the correct metadata locations in each event and writes the latency statistics to the configured output file.

To measure model and network latencies by modifying your publish/subscribe clients:

1 In the model, call the `dfESPengine setLatencyMode()` function before starting any projects.

2 In your publisher client application, immediately before calling `C_dfESPpublisherInject()`, call `C_dfESPlibrary_getMicroTS()` to get a current timestamp. Loop through all events in the event block and for each one call `C_dfESPevent_setMeta(event, 0, timestamp)` to write the timestamp to the event. This records the publish/subscribe inject timestamp to meta location 0.

3 The model inject and subscriber callback timestamps are recorded to meta locations 2 and 3 in all events automatically because latency mode is enabled in the engine.

4 Add code to the inject loop to implement a fixed inject rate. See the latency publish/subscribe client example for sample rate limiting code.

5 In your subscriber client application, include "`int/dfESPlatencyController.h`" and add an instance of the `dfESPlatencyController` object.

6 Configure the latency controller bucketSize and playbackRate parameters as described previously.

7 Pass your latency controller object as the context to `C_dfESPsubscriberStart()` so that your subscriber callback has access to the latency controller.

8 Make the subscriber callback pass the latency controller to `C_dfESPlatencyController_recordOutputEvents()`, along with the event block. This records the publish/subscribe callback timestamp to meta location 4.

9 When the subscriber client application has received all events, you can generate statistics for latencies between any pair of the four timestamps recorded in each event. First call `C_dfESPlatencyController_setOFile()` to set the output file. Then write the statistics to the file by calling `C_dfESPlatencyController_generateStats()` and passing the latency controller and the two timestamps of interest. The list of possible timestamp pairs and their time spans are as follows:

- (0, 2) – from inject by the publisher client to inject by the model
- (0, 3) – from inject by the publisher client to subscriber callback by the model
- (0, 4) – from inject by the publisher client to callback by the subscriber client (full path)
- (2, 3) – from inject by the model to subscriber callback by the model
- (2, 4) – from inject by the model to callback by the subscriber client
- (3, 4) – from subscriber callback by the model to callback by the subscriber client

10 To generate further statistics for other pairs of timestamps, reset the output file and call `C_dfESPlatencyController_generateStats()` again.

To measure model and network latencies by using the file/socket adapter, run the publisher and subscriber adapters as normal but with these additional switches:

Publisher	
—r *rate*	Specifies the requested transmit rate in events per second.
-m *maxevents*	Specifies the maximum number of events to publish.
-p	Specifies to buffer all events prior to publishing.
-n	Enables latency mode.

Subscriber	
-r *rate*	Specifies the requested transmit rate in events per second.
-a *aggrsize* <*aggr_blocks_to_ignore*>	Specifies the aggregation bucket size. Can be followed by a comma-separated value that specifies the beginning and ending aggregation blocks to ignore in latency calculations.
-n	Enables latency mode.

	Subscriber
-N *latencyblksize*	Specifies the block size (in number of events) of memory to allocate for storing timestamps when in latency mode. Additional blocks are allocated as required.

The subscriber adapter gathers all four timestamps described earlier for the windows specified in the respective publisher and subscriber adapter URLs. At the end of the run, it writes the statistics data to files in the current directory. These files are named "latency_**`transmit rate_high timestamp_low timestamp`**", where the high and low timestamps correspond to the timestamp pairs listed earlier.

Enabling Finalized Callback

Some data structures are fully created when windows and edges are made, but are finalized just before the project is started. These data structures include derived schema and certain types of window indexes. The finalized callback function is called when all data structures are completely initialized, but before any events start to flow into the window. The finalized callback function can initialize some state or connection information that is required by an application or XML model.

Enable finalized callback as follows:

- Use the finalized-callback element in XML. Specify the name of the library that contains the window callback function and the name of the function that the window calls.

  ```
  <finalized-callback name='library' function='fin_callback'>
  ```

- Use the following function in C++:
  `dfESPwindow::addFinalizeCallback(dfESPwindowCB_func` *`cbf`*`)`

Using Code-Based Routing

Overview

Code-based routing enables you to configure communication between ESP servers by defining routes within a project model. Multiple routes are allowed. Each route consists of independently defined objects (for example, connections, sources, destinations, routers, and modifiers) that are connected using edges. For more information, see "XML Language Elements for the Structure of a Model" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Stock Trade Example

This example uses stock trade data. It consists of three models called source.xml, exchanges.xml, and sectors.xml. The input data file was created by GitHub Copilot and contains 98 of the most active symbols:

```
{
    rank = 1,
    symbol = "NVDA",
    company = "NVIDIA Corporation Common Stock",
    exchange = "NASDAQ",
    sector = "Technology",
    min_trade_price = 138.83,
    max_trade_price = 236.54,
    min_trade_quantity_est = 161622849,
    max_trade_quantity_est = 162981498,
  },
  {
    rank = 2,
    symbol = "AAPL",
    company = "Apple Inc. Common Stock",
    exchange = "NASDAQ",
    sector = "Technology",
    min_trade_price = 195.07,
    max_trade_price = 316.94,
    min_trade_quantity_est = 44311442,
    max_trade_quantity_est = 45325667,
  },
  ...
```

The source.xml model contains a Lua connector that generates events and route definitions. The connector reads the input data and generates random entries from the file to match the Source window schema:

```
<window-source name="trades" insert-only="true"> 1
  <schema>
    <fields> 2
      <field key="true" name="id" type="int64"/>
      <field name="symbol" type="string"/>
      <field name="company" type="string"/>
      <field name="sector" type="string"/>
      <field name="exchange" type="string"/>
      <field name="price" type="double"/>
      <field name="quantity" type="int32"/>
      <field name="broker" type="string"/>
      <field name="buyer" type="string"/>
      <field name="seller" type="string"/>
    </fields>
  </schema>
</window-source>
```

1 This line defines a Source window called trades that accepts insert events only.

2 Several event fields are defined using names and types.

The exchanges.xml model consists of two continuous queries, one for NYSE and one for NASDAQ. The input Source window is the same as the window for source.xml with the addition of an event field called `tradevalue` with type **`int64`**.

The sectors.xml model also consists of two continuous queries, one for focus sectors and one for other sectors. The input Source window is the same as the window for exchanges.xml.

This example uses a source model that generates stock trade data that is pushed into Source windows that are in two other servers. The router configuration is shown here:

```
<routes>
  <route name="traderoute">
    <connections> 1
      <connection name="exchangesserver" esp-project="exchanges"
url="http://espsrv04:3333"/>
      <connection name="sectorsserver" esp-project="sectors"
url="http://espsrv04:4444"/>
    </connections>
    <sources>
      <source name="src" window="trades"/> 2
    </sources>
    <destinations> 3
      <destination name="NYSE" connection="exchanges" window="NYSE/
src"/>
      <destination name="NASDAQ" connection="exchanges" window="NASDAQ/
src"/>
      <destination name="focusedSectors" connection="sectors"
window="focused/src"/>
      <destination name="otherSectors" connection="sectors"
window="other/src"/>
    </destinations>
    <routers>
      <router name="trades" func="routeTrade" use="exchange,sector"/>
4
    </routers>
    <modifiers>
      <modifier name="tradeValue" func="addTradeValue"
use="quantity,price"/>
    </modifiers>
    <edges>
      <edge source="src://src" target="router://trades" /> 5
      <edge source="dest://*" target="mod://tradeValue" /> 6
    </edges>
    <code><![CDATA[

      function addTradeValue(event)
          local   data = {}
          data.tradevalue = event.quantity * event.price
          return data
      end

      function routeTrade(event)

          local   dests = {}
```

```
            dests[#dests + 1] = event.exchange

            if (event.sector == "Technology" or event.sector == "Health
Care")
            then
              dests[#dests + 1] = "focusedSectors"
            else
              dests[#dests + 1] = "otherSectors"
            end

            return dests
        end

    ]]></code>
  </route>
</routes>
```

1 The connections element defines the other two ESP servers. They are called `exchangesserver` and `sectorsserver`.

2 The source element defines the single Source window to receive events from.

3 The destinations element defines four destinations. Each server has two destinations. The destinations point to different continuous queries in the respective servers.

4 The router element defines a single router to route events using the routeTrade function. The function uses the exchange event field to route the event to the matching destination, which is either NYSE or NASDAQ. The sector event field is used to route the event to either the `focusedSectors` or `otherSectors` destination. Because exchange and sector are the only event fields required for this routing, they are specifies in the use attribute.

5 This edge takes elements from the `src` source and sends them through the trades router, which returns the appropriate destinations.

6 This edge applies modifications to all events. The modifications are defined in the `tradeValue` modifier.

PART 2

Functionality Common to Multiple Windows

8

Common Lua Functionality

User-Provided Code in ESP Projects

SAS Event Stream Processing enables you to extend your projects with your own code (for example, Python, Lua, or other custom components). You and your organization are solely responsible for the behavior and security of any code you decide to use. SAS Event Stream Processing does not review or test user-provided code for security purposes. Any code or other customer-supplied materials you add to an ESP project remain your responsibility.

CAUTION
Do not run unsafe code. Running unsafe code may expose your environment to data loss, data corruption, security breaches, or system instability. It is your responsibility to ensure that all code you incorporate is secure, tested, and compliant with your organization's policies and applicable law.

Lua Programming Concepts

Lua is a dynamically typed programming language. It is designed to be lightweight, embeddable, and well suited for writing plug-in programs. Lua is embedded into the ESP server, so no external Lua interpreter is required to run Lua code within any of the Lua entities that you can create within a project. For more information about the Lua language, see the Lua documentation.

IMPORTANT Your Lua code can include any function provided in the standard Lua libraries with the following two exceptions:

- `os.execute`
- `io.popen`

The use of either of these functions results in a runtime error, which prevents a project from being loaded.

Tables provide the primary data structure in Lua. Tables are objects that you access by key or by index. You can use Lua tables to convert SAS Event Stream Processing objects into Lua objects and vice versa. You can work with Lua objects similarly to how you work with JavaScript objects.

Consider the following Lua table named `employee`. There are two fields:

Table 8.1 Lua Table

Key	Value
name	Jose
age	30

You can reference the key in brackets.

```
local employee = {}
employee["name"] = "Jose"
employee["age"] = 30
```

You can also use dot notation to reference the key:

```
local employee = {}
employee.name = "Jose"
employee.age = 30
```

The following Lua code iterates over the values of the previous table:

```
for key,value in pairs(employee)
do
    print(key,"=",esp_toString(value))
end
```

The following Lua table uses an indexed array to access field values:

Note: Lua indexed arrays start at index 1, not index 0.

```
local names = {}
o[1] = "Jose"
o[2] = "Sally"
o[3] = "Miko"
```

The following Lua code iterates over the values of that table:

```
for index,value in ipairs(names)
do
    print(esp_toString(index),"=",value)
end
```

Initialization of Lua Entities

Initialization is the process of locating and using defined values for variable data. An initialization mechanism is available to all Lua entities in a SAS Event Stream Processing project when the ESP server starts up. Different Lua entities initialize in different ways.

The following Lua entities implement the initialization mechanism through an `init` attribute in the corresponding XML element:

- Lua window (window-lua)
- Lua-based Pattern window (window-pattern; the `init` attribute is applied to each pattern)
- Lua-based Filter window (window-filter)

The Lua connector entity provides an optional `init` parameter that points to an initialization function in the Lua code.

Initialize a Lua splitter entity by including a function called `init` in the Lua code.

When the ESP server starts, actions that are related to the processing of Lua code are performed in the following sequence:

1. Loads persistent properties for Lua snippets.
2. Loads persistent properties for Lua windows.
3. For any Lua snippet that includes one, calls the `init` function.
4. Starts any threaded Lua snippets.
5. For any Lua window, Lua-based Pattern window, or Lua-based Filter window that includes one, calls the `init` function. When the window contains a Lua splitter that includes one, calls the `init` function.

All of the preceding operations are performed in the order in which Lua entities are defined in the project.

It is recommended to use initialization functions whenever you have Lua entities that get property values from other Lua entities. For example, consider the following code:

```
local myvalue = esp_getProperty({name="value"})
```

If no previous Lua entity had set the **`value`** property when the code was called, the code produces a **`nil`** value. When you put the code in an initialization function, you can be sure that the property has been set when any another Lua entity has been initialized:

```
local   myvalue = nil

function init()
    myvalue = esp_getProperty({name="value"})
end
```

Using SAS Event Stream Processing Functions with Lua

Overview

The ESP server provides functions that you can invoke within Lua code that enable the following activities.

Table 8.2 *Functions Organized by Activity*

Activity	Functions

Working with events	`esp_publish`
	`esp_publishAsynch`
	`esp_query`
	`esp_subscribe`
	`esp_unsubscribe`
	`esp_waitForEvent`

Parsing and encoding JSON	esp_jsonEncoder esp_parseJson esp_parseJsonFrom
Parsing XML	`esp_parseXml` `esp_parseXmlFrom`

Working with regular expressions	`esp_getExpressionValues`
	`esp_getExpressionValuesFrom`
	`esp_setExpression`

Working with HTTP requests	`esp_sendHttp`
Generating GUIDs	`esp_getGuids`
Debugging	`esp_logMessage`
Directly injecting CSV data	esp_injectCsv

Activity	Functions
Miscellaneous functions	esp_getUTCOffset esp_getUTCOffsetHours esp_inlist esp_isDone esp_toString esp_getServerProperty esp_getProjectProperty
String function	esp_strings
Event context functions	esp_getEventFlags esp_getEventOpcode
Date and time functions	esp_dateMath esp_formatDate esp_formatTime esp_getSystemMicro esp_getSystemMilli esp_parseDate
Working with Lua properties on page 275	esp_setProperty esp_getProperty esp_clearProperty esp_addPropertyListener
Accessing Relational Databases	esp_sqlConnect esp_sqlDisconnect esp_sqlSelect esp_sqlOpen esp_sqlNext esp_sqlClose esp_sqlInsert esp_sqlUpdate esp_sqlDelete

IMPORTANT In releases before 2022.10, functions did not include the esp_ prefix. For example, the esp_parseJson function was parseJson. For many

functions, parameters have changed since 2022.10. Be aware of these differences as you migrate projects from releases earlier than 2022.10.

Working with Events

Publishing Events

Use the `esp_publish` or `esp_publishAsynch` functions to publish events to Source windows. This function can be useful when you want to send event data down an event stream different from the current one.

Table 8.3 *Parameters of the esp_publish and esp_publishAsynch Functions*

Parameter	Description	
`window`	Specifies the name of the Source window. Use one of the following forms: ■ *project/query/window* ■ *query/window* ■ *window*	
`events`	Specifies the event data to publish (either a single Lua object or an array of objects).	
`opts`	Specifies one of the following publishing options:	
	`logs=true | false`	When true, the ESP server logs are collected during publishing and returned in the response (only valid for `esp_publish`).
	`blocksize`	Specifies the publish *blocksize*. The default value is 1.

For example:

```
local response = esp_publish({window="query/
window",events=events,opts={logs=true}})
local response = esp_publishAsync({window="query/
window",events=events})
```

The esp_publish function injects events into the query before it returns a value. The esp_publishAsync function puts events into a queue to be processed asynchronously. The return value contains a code field and a message field. When you specify the logs parameter, the server logs are returned in the logs field.

By default, the Insert opcode is used to publish events. When you set the value of the _opcode field of a Lua object or set of objects, that value is used to set the opcode.

For example, suppose that you want to process stock trade data and want to send broker statistics into their own Source window. The following Lua window receives trade events from the event stream. It first uses the name of the broker (brokerName) to store the trade statistics for each broker (price, quant). It then performs computations to determine maximums and averages for trade quantity and price (stats.max_x, stats.avg_x).

```
<window-lua name="publishStats" events="f">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
    </fields>
  </schema>
  <use>brokerName,price,quant</use>
  <code><![CDATA[

    local   brokerdata = {}

    function f(data)
        local   stats = brokerdata[data.brokerName]

        if stats == nil
        then
            stats =
{broker=data.brokerName,numtrades=1,max_price=data.price,avg_price=data
.price,max_quant=data.quant,avg_quant=data.quant,total_price=data.price
,total_quant=data.quant}
            stats._opcode = "upsert"
            brokerdata[data.brokerName] = stats
        else
            stats.numtrades = stats.numtrades + 1
            stats.total_price = stats.total_price + data.price
            stats.total_quant = stats.total_quant + data.quant

            if data.price > stats.max_price
            then
                stats.max_price = data.price
            end

            if data.quant > stats.max_quant
            then
                stats.max_quant = data.quant
            end

            stats.avg_price = stats.total_price / stats.numtrades
            stats.avg_quant = stats.total_quant / stats.numtrades
        end
```

```
            esp_publish("brokerstats",stats)

            return nil
        end

    ]]></code>
</window-lua>
```

Note that the event creation function returns `nil` so that no event is created in its own stream. The function only publishes events into another path.

The Lua window publishes the statistics into the `brokerstats` window in the same continuous query:

```
<window-source name="brokerstats" index="pi_HASH">
  <schema>
    <fields>
      <field name="broker" type="string" key="true"/>
      <field name="numtrades" type="int32"/>
      <field name="max_price" type="double"/>
      <field name="avg_price" type="double"/>
      <field name="max_quant" type="double"/>
      <field name="avg_quant" type="double"/>
    </fields>
  </schema>
</window-source>
```

Subscribing to Events

Use the `esp_subscribe` function to subscribe to events. This function takes a single Lua object as a parameter.

Table 8.4 *Parameters of the esp_subscribe Function*

Parameter	Description	
`window`	Specifies the window from which to receive events.	
`handler`	Specifies the name of the Lua function that is used to receive events.	
`output`	Specifies one of the following output options:	
	`fields`	Specifies a comma-separated list of fields to include in event delivery.
	`exclude=true | false`	Specifies whether the items that are specified in *`fields`* should be included

Parameter	Description	
		or excluded. The default value is **`false`**.
	`encode_binary=true | false`	Specifies whether to use Base64 to encode binary data for event delivery. The default value is **`false`**.

For example:

```
local subscriber = esp_subscribe({window="project/query/
window",handler="handler"}) function handler(events) print("received
some events: "..esp_toString(events)) end
```

Because subscribing to events creates a subscription resource in the server, you must clean up this resource when it is no longer needed. Use the `esp_unsubscribe` function:

```
function done()
     esp_unsubscribe(subscriber)
end
```

The `subscriber` parameter is the value that was returned by the `esp_subscribe` function call.

Here is an example of a Lua snippet that subscribes to one window and publishes into another:

```
<lua-snippet name="snippet" init="init" destroy="done">
    <code><![CDATA[

        local   subscriber = nil

        function init()
            subscriber = esp_subscribe({window="p/cq/
alert",handler="handler"})
        end

        function handler(events)
            esp_publish({window="cq2/
storeAlerts",events=events,opts={logs=true}})
        end

        function done()
            esp_unsubscribe(subscriber)
        end

    ]]></code>
</lua-snippet>
```

Querying Events

Use the `esp_query` function to query events. This function takes a single Lua object as a parameter.

Table 8.5 *Parameters of the esp_query Function*

Parameter	Description	
`window`	Specifies the window from which to receive events.	
`output`	Specifies one of the following output options:	
	`fields`	Specifies a comma-separated list of fields to include in event delivery.
	`exclude=true | false`	Specifies whether the items that are specified in `fields` should be included or excluded. The default value is **`false`**.
	`encode_binary=true | false`	Specifies whether to use Base64 to encode binary data for event delivery. The default value is **`false`**.
`filter`	Specifies one of the following filter options:	
	`func`	Specifies the name of the function that is used to filter events. Write code so that the function takes an event and returns `true` when the event matches the desired criteria. Otherwise, make the function return `false`.
	`use`	Use one of the following options to specify what fields to send to the filter function: ■ `fields`: Specifies a comma-separated list

Parameter	Description
	of fields on which to sort. ■ `dir`: Specifies the sort direction, *ascending* or *descending* (default). ■ `maxevents` Specifies the maximum number of events to return. By default, all events are returned.

For example:

```
local   patients = esp_query({window="patients"})
local   vitals = esp_query({window="vitals"})
```

Suppose that you had a model that processes stock trades and you wanted to get the top 10 trades by broker 'Joe', sorted by quantity:

```
local   events = esp_query({window="p/cq/trades",
                            filter={func="filter",use="broker"},
                            output={fields="symbol,quantity,price"},
                            sort={fields="quantity",maxevents=10})

print("events\n"..esp_toString(events))

function filter(event)
    return event.broker == "Joe"
end
```

Waiting for Events

Use the `esp_waitForEvent` function to wait for an event to transpire before processing proceeds.

Table 8.6 *Parameters of the esp_waitForEvent Function*

Parameter	Description
`window`	Specifies the window from which to receive events.
`match`	Specifies the name of the function that is used to determine whether an event is a match. This parameter returns **`true`** or **`false`**.
`timeout`	Specifies the amount of time to wait for the event to occur. Use units such as **`'10 seconds'`** or **`'1 minute'`** This parameter defaults to no time-out.

Parameter	Description	
`send_update_deletes=true | false`	Specifies whether to send delete events to the match function for an update block. The default value is **`false`**.	
`snapshot=true | false`	Specifies whether to process the current contents of the window. When **`false`** (the default value), the function processes only future events.	
`use`	Specifies how to handle event fields that are sent to the match function.	
	`fields`	Specifies a comma-separated list of fields to include in an event that is sent to the match function.
	`exclude=true | false`	Specifies whether items in fields should be included or excluded. The default value is **`false`**.
	`encode_binary=true | false`	Specifies whether to use Base64 to encode binary data for event delivery. The default value is **`false`**.
`output`	Specifies how to handle event fields to include in the output event.	
	`fields`	Specifies a comma-separated list of fields to include in an event that is sent to the match function.
	`exclude=true | false`	Specifies whether the items in fields should be excluded. The default value is **`false`**.
	`encode_binary=true | false`	Specifies whether to use Base64 to encode binary data for event delivery. The default value is **`false`**.

For example:

```
local   event,err = esp_waitForEvent({window="p/cq/brokerAlertsAggr",
```

```
                                        match="match",
                                        timeout="5 seconds",
                                        snapshot=true})

function match(event)
    return event.value > 10
end
```

The function returns the desired event if it occurs. Otherwise, the function returns nil or error. If there is an error, then a second return value includes a message that indicates the reason for failure.

Suppose that you have a project that analyzes stock trades. You want to wait for a trader named `'Joe'` to amass 10 trade violations before taking action. You can set up a Lua snippet to perform this function:

```
<lua-snippet name="snippet">
    <lua-thread func="dowait" interval="1 second"/>
    <code><![CDATA[

        function dowait()
            io.write("running dowait, waiting for event...")
            io.flush()

            local   event,err = esp_waitForEvent({window="p/cq/
brokerAlertsAggr",
                                                  match="match",
                                                  timeout="5
seconds",

use={fields="brokerName,total"},

output={fields="frontRunningBuy",exclude=true},
                                                  snapshot=true})

            if (err ~= nil)
            then
                io.write("failed to get event, "..err.."\n")
            else
                io.write("got event, dowait continuing\n")
                io.write(esp_toString(event))
            end

            return true
        end

        function match(event)
            return event.brokerName == "Joe" and event.total > 10
        end

    ]]></code>
</lua-snippet>
```

Generating GUIDs during Event Generation

The `esp_getGuids` function enables you to generate globally unique identifiers (GUIDs) when you generate events.

Table 8.7 *esp_getGuids Function*

Function	Description
`esp_getGuids(`*`numguids`*`)`	Retrieves *numguids* GUIDs from the ESP server. The GUIDs are returned to Lua in an indexed array. The default value of *numguids* is 1.

For example, the following code generates five GUIDs:

```
local   guids = esp_getGuids(5)

for index,guid in ipairs(guids) do
    -- Generate event with GUID
end
```

Parsing JSON

JSON transmits data in attribute-value pairs. It is widely used with real-time stateless communication protocols.

The following functions enable you to parse JSON within Lua code.

Table 8.8 *JSON Parsing Functions*

Function	Description
`esp_parseJson(`*`stringdata`*`)`	Sends string data from Lua into the ESP server, which is then parsed and returned as a Lua object.
`esp_parseJsonFrom("`*`stringdata`*`")`	Sends string data from the ESP server into Lua. The string must specify the name of a field in an input event, which is then parsed and returned as a Lua object.

Note: Use this function only when an input event is processed.

Suppose that you had an event field named `sensorinfo` with sensor data in JSON format. Specify the call to `esp_parseJson` as follows:

```
local   parsed = esp_parseJson(data.sensorinfo)
```

Object notation is used because the function is parsing a field that is being passed in event data. The `esp_parseJson` function is a general-purpose utility that you can use on any data, not only event data.

Specify the call for `esp_parseJsonFrom` as follows:

```
local   parsed = esp_parseJsonFrom("sensorinfo")
```

The `esp_parseJsonFrom` function does not require the data to be passed to the Lua code because the call is picking up the data from the ESP server.

Suppose that the following input data (in JSON format) arrived in an event field called `sensorinfo`:

```
[
    {
        device-id:1000,
        sensor-id:10,
        value:1.54
    },
    {
        device-id:1000,
        sensor-id:20,
        value:9.23
    },
    {
        device-id:2000,
        sensor-id:10,
        value:3.24
    }
]
```

Suppose that you want to create events from this JSON data. Your Lua window would be as follows:

```
<window-lua name="lua" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="device_id" type="string"/>
      <field name="sensor_id" type="string"/>
      <field name="value" type="double"/>
    </fields>
  </schema>
  <use/>
  <code><![CDATA[

    local   eventId = 1

    function create(data,context)

        local   events = {}
        local   sensorinfo = esp_parseJsonFrom("sensorinfo")
```

```
        for index,value in ipairs(sensorinfo)
        do
            local   e = {}

            e.id = esp_toString(eventId)
            e.device_id = value.device_id
            e.sensor_id = value.sensor_id
            e.value = value.value
            events[index] = e

            eventId = eventId + 1
        end

        return(events)
    end

  ]]></code>
</window-lua>
```

For an example, see Generate Events from JSON Using the Lua Window.

Encoding JSON

Use the `esp_jsonEncoder(`*object*`,`*format*`)` function to encode a Lua object as a JSON string.

Table 8.9 *Parameters of the esp_jsonEncoder Function*

Parameter	Description
object	Specifies the name of the Lua object that you want to convert to JSON.
format	Is a Boolean value that specifies whether to format (pretty print) the JSON. The default value is **`false`**.

Consider the following event creation function:

```
function create(data,context)
    local   e = {}
    e.id = 1
    local   afc = {"East","North","South","West"}
    local   nfc = {"East","North","South","West"}
    local   nfl = {nfl={divisions={afc,nfc}}}
    e.league = esp_jsonEncoder(nfl)
    return e
end
```

The function creates an event that looks like this:

```
<event opcode="insert" window="p/cq/code">
```

```
  <value name="id">1</value>
  <value name="league">
{
    "nfl":{
        "divisions":[
            [
                "East",
                "North",
                "South",
                "West"
            ],
            [
                "East",
                "North",
                "South",
                "West"
            ]
        ]
    }
}
</value>
</event>
```

Parsing XML

The following functions enable you to parse XML in the Lua window.

Table 8.10 *XML Parsing Functions*

Function	Description
`esp_parseXml(`*`stringdata`*`)`	Sends string data from Lua into the ESP server, which is then parsed and returned as a Lua object.
`esp_parseXmlFrom("`*`stringdata`*`")`	Sends string data from the ESP server into Lua. The string could specify the name of a field in an input event, which is then parsed and returned as a Lua object.

Suppose that you had an event field called `sensorinfo` with sensor data in XML format. The call for `esp_parseXml` would be as follows:

```
local parsed = esp_parseXml(data.sensorinfo)
```

The `esp_parseXml` function is a general-purpose utility that can be used on any data, not only event data.

The call for `esp_parseXmlFrom` would be as follows:

```
local parsed = esp_parseXmlFrom("sensorinfo")
```

Suppose that the following sensor data (in XML format) arrived in a field called sensorinfo:

```
<data>
    <sensorinfo>
        <device_id>1000</device_id>
        <sensor_id>10</sensor_id>
        <value>1.54</value>
    </sensorinfo>
    <sensorinfo>
        <device_id>1000</device_id>
        <sensor_id>20</sensor_id>
        <value>9.23</value>
    </sensorinfo>
    <sensorinfo>
        <device_id>2000</device_id>
        <sensor_id>10</sensor_id>
        <value>3.24</value>
    </sensorinfo>
</data>
```

If you wanted to create events from this XML, you could specify a Lua window as follows:

```
<window-lua name="lua" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="device_id" type="string"/>
      <field name="sensor_id" type="string"/>
      <field name="value" type="double"/>
    </fields>
  </schema>
  <use/>
  <code><![CDATA[

    local   eventId = 1

    function create(data,context)

        local   events = {}
        local   sensorinfo = esp_parseXmlFrom("sensorinfo")
        print(esp_toString(sensorinfo))

        for index,value in ipairs(sensorinfo[1].data)
        do
            local   e = {}

            e.id = esp_toString(eventId)
            e.device_id = value.sensorinfo.device_id
            e.sensor_id = value.sensorinfo.sensor_id
            e.value = value.sensorinfo.value
            events[index] = e

            eventId = eventId + 1
        end

        return(events)
```

```
    end

  ]]></code>
</window-lua>
```

Note: When XML data does not map intuitively into Lua objects, you can use `print(esp_toString(sensorinfo))` to expose the structure of the data. Then, you can traverse that structure in order to retrieve the data.

Working with Regular Expressions

The ESP server provides the capability to evaluate regular expressions through the `esp_setExpression` function.

```
esp_setExpression(name,regex,ignorecase)
```

This function stores a compiled version of the regular expression (*regex*) in the ESP server, which can then be accessed by *name*.

The `ignorecase` parameter is optional. Set the parameter to **`true`** when you want the expression to be case -insensitive. The default value of `ignorecase` is **`false`**.

Then, you can reference the expressions by using these functions:

- `esp_getExpressionValues(expr,text)`, where *`expr`* specifies the name of the predefined expression and *`text`* specifies the text field to which the expression can be applied
- `esp_getExpressionValuesFrom(expr,field)`, where *`expr`* is the name of the predefined expression and *`field`* is the name of the input field

Note: Use `esp_getExpressionValuesFrom` only when processing an input event.

In the following code, `esp_setExpression` compiles and stores the regular expression `\w+` in the ESP server. The code then applies the expression, using the name **`words`**, to text values through the `esp_getExpressionValues` function in order to detect matches.

```
esp_setExpression("words","\\w+")

local   words,count = esp_getExpressionValues("words","This is a test")

for i=1,count do
    print("word is: "..words[i])
end
```

Suppose that you had input in paragraph format and you wanted to break up the paragraph into individual sentences. You could use the two functions as follows:

```
<window-lua name="code" events="create">
    <schema>
        <fields>
            <field name="id" type="int32" key="true"/>
```

```
            <field name="sentence" type="string"/>
        </fields>
    </schema>
    <use/>
    <code><![CDATA[

        esp_setExpression("sentences","[A-z0-9 ]*\\.")

        local   id = 1

        function create(data,context)
            local   sentences, count =
esp_getExpressionValuesFrom("sentences","msg")
            local   events = {}

            for i=1,count
            do
                local   sentence = sentences[i]

                if (string.len(sentence) > 5) then
                    local   event = {}
                    event.id = id
                    event.sentence = sentence
                    events[i] = event
                    id = id + 1
                end
            end

            return(events)
        end

    ]]></code>
</window-lua>
```

When the input field `msg` contains text in a paragraph, the Lua code calls back into the ESP server to break the paragraph into sentences that are used to create individual events.

Sending HTTP Requests and Receiving Responses

You can send HTTP requests, and then use the `esp_sendHttp(`*`fields`*`)` function to direct responses back into the Lua code of a Lua-based window. Format a request object and pass it to the function with the following *fields*:

Table 8.11 *Request Object Fields*

Field	Description
`url`	Specifies the URL of the request.

Field	Description
`method`	Specifies the HTTP method to use (**`GET`**, **`PUT`**, **`POST`**, **`DELETE`**). The default value is **`GET`**.
`headers`	Specifies HTTP request headers to send to the server.
`body`	Specifies the body of the HTTP request for **`PUTs`** and **`POSTs`**.
`tolua`	When the HTTP response is in JSON or XML, you can set this field to **`true`** in order for the ESP server to convert the response into a Lua table before sending the response back.
`connect-timeout`	Specifies the time-out in number of seconds to wait before successfully establishing a connection with the server. The default value is `0` (no time-out).
`data-timeout`	Specifies the time-out in number of seconds to wait before successfully receiving data from the server. The default value is `0` (no time-out).

For example, the following code requests information from the OpenWeather Weather API:

```
local   request = {}

request["url"] = "http://api.openweathermap.org/data/2.5/group?
id=4487042&APPID=d0b38229a8f66dac4fbce8bf76be97b4&units=imperial"
request["method"] = "GET"
request["headers"] = {accept="application/json"}

local   response = esp_sendHttp(request)

print(esp_toString(response))
```

That code elicits a response that is similar to the following code from the API:

```
status=200
response={"cnt":1,"list":[{"coord":{"lon":-78.6386,"lat":35.7721},
"sys":{"country":"US","timezone":-14400,"sunrise":1650450904,"sunset":1650498685},
"weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}],
"main":
{"temp":40.91,"feels_like":40.91,"temp_min":35.08,"temp_max":46.42,"pressure":1031
,"humidity":89},
"visibility":10000,"wind":{"speed":0,"deg":0},"clouds":{"all":20},
"dt":1650456592,"id":4487042,"name":"Raleigh"}]}
```

When you set the `tolua` field to **`true`**, the object returns a ready-to-use Lua table. For example, consider the following code:

```
local   request = {}

request["url"] = "http://api.openweathermap.org/data/2.5/group?
id=4487042&APPID=d0b38229a8f66dac4fbce8bf76be97b4&units=imperial"
```

```
request["method"] = "GET"
request["headers"] = {accept="application/json"}
request["tolua"] = true

local   response = esp_sendHttp(request)

print(esp_toString(response))
```

That code returns something that is similar to the following example:

```
status=200
response=
    list=
        1=
            clouds=
                all=20
            main=
                temp_max=46.42
                temp_min=35.08
                pressure=1031
                temp=40.91
                humidity=89
                feels_like=40.91
            visibility=10000
            dt=1650456592
            weather=
                1=
                    id=801
                    description=few clouds
                    icon=02d
                    main=Clouds
            sys=
                sunset=1650498685
                timezone=-14400
                sunrise=1650450904
                country=US
            coord=
                lon=-78.6386
                lat=35.7721
            wind=
                speed=0
                deg=0
            name=Raleigh
            id=4487042
    cnt=1
```

Debugging

Overview

You can add diagnostic output directly into Lua functions. There are two ways to view this output:

- through the console
- through the ESP server log

Console Debugging

You can use the Lua `print()` function to print diagnostic output to the console. This capability is useful when you develop in a command-line environment and run models directly in a console.

For example, if you are looking at sequences in stock trade information, you could dump the current event along with the context when a function runs. At the end of the function, you can dump the return value:

```
function f3(event,context)

        print("in function e3")
        print(esp_toString(event))
        print(esp_toString(context))

        local   code = event.broker == "George"
                and event.price < context.events.john.price
                and event.price < context.events.paul.price

        print("returning: "..code.." from f3")

        return code
    end
```

When the previous model is run, you see output that is similar to this example:

```
in function e3

        price=50.0
        quantity=50
        broker=George
        id=6
        symbol=IBM


        name=george
        window=1
        events=
            john=
                price=97.34
                quantity=500
                broker=John
                id=0
                symbol=IBM
            paul=
                price=57.34
                quantity=500
                broker=Paul
                id=3
                symbol=IBM

    returning: true from f3
```

When you dump the events and context, you can follow the patterns and determine why certain events trigger certain patterns.

ESP Server Log Debugging

You can use the `esp_logMessage` function to log messages from Lua to the ESP server log. The `esp_logMessage` function takes the following parameters:

Table 8.12 *Parameters for the esp_logMessage Function*

Parameter	Description
`logcontext`	Specifies the context object in the ESP server (for example, **`DF.ESP`**). This parameter is required.
`message`	Specifies the message that you want to log. This parameter is required.
`level`	Specifies the log level to use. If the specified log context is set to that log level or higher, the message is logged. Otherwise, the message is discarded. Possible values of `level` are as follows: ■ **`trace`** ■ **`error`** ■ **`warn`** ■ **`fatal`** ■ **`info`** ■ **`debug`** This parameter is optional and defaults to **`info`**.
`line`	Specifies the line number of the Lua code from which the message is logged. This value is generated with the following Lua call: `debug.getinfo(1).currentline` If you do not specify a value for `line`, no Lua line number is logged.

Consider the following function that receives an event and generates an event with some minor data modifications. The code logs the incoming event and the generated event if the `"mylogger"` logging context is set to **`debug`** or higher:

```
local   eventId = 1

function create(data,context)
```

```
    esp_logMessage("mylogger","creating event
from"..esp_toString(data),"debug",debug.getinfo(1).currentline)

    local   e = {}

    e.id = esp_toString(eventId)
    e.new_string = "\""..data.string.."\""
    e.new_number = data.number * 3
    e.new_array = {}
    for index,value in ipairs(data.array) do
        e.new_array[index] = value * 2
    end

    eventId = eventId + 1

    esp_logMessage("mylogger","created
event"..esp_toString(e),"debug",debug.getinfo(1).currentline)

    return(e)
end
```

The log output might look similar to this example:

```
_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] creating
event from
    id=1
    string=string data
    array=
        1=11
        2=24
        3=55
    number=33
 (Lua line 7)
messageFile=./src/dfESPwindow_lua.cpp
messageLine=774

_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] created event
    id=1
    new_string="string data"
    new_array=
        1=22
        2=48
        3=110
    new_number=99
 (Lua line 21)
messageFile=./src/dfESPwindow_lua.cpp
messageLine=774
```

To debug pattern matching in a pattern window, you could write messages to the ESP server log by using the following code:

```
function f3(event,context)

    local   message = "in function e3"
```

```
    message = message.."\n"..esp_toString(event)
    message = message.."\n"..esp_toString(context)

    local   code = event.broker == "George"
            and event.price < context.events.john.price
            and event.price < context.events.paul.price

    message = message.."\n".."returning: "..esp_toString(code).." from
f3"


esp_logMessage("mypatternlogger",logmessage,"debug",debug.getinfo(1).cu
rrentline)

    return code
end
```

This code matches log entries where `level=debug`. Assuming that you set the logging context appropriately, you see something that is similar to the following example in the ESP server log:

```
{
  "_timestamp": "2021-12-16T12:36:43.312000-05:00",
  "logger": "mypatternlogger",
  "loggerLevel": "DEBUG",
  "messageContent": "{f3b6119e-e65c-4ad9-a09b-36075fc43339}[XMLServer0001] in
function e3\n\n    symbol=IBM\n    quantity=50\n    id=6\n    broker=George\n
price=50.0\n\n\n    window=1\n    name=george\n    events=\n        paul=
\n            symbol=IBM\n            quantity=500\n            id=3\n
broker=Paul\n            price=57.34\n        john=\n            symbol=IBM
\n            quantity=500\n            id=0\n            broker=John
\n            price=97.34\n\nreturning: true from f3 (Lua line 23)",
  "messageFile": "dfESPwindow_luabase.cpp",
  "messageLine": "130",
  "thread": "00000034"
}
```

You can get the current log level of loggers in the ESP server with the `esp_getLogLevel()` function. This function returns information about an individual logging context or all contexts if no name is specified.

Both the numeric level and level name are returned in the data.

Table 8.13 *Numeric Levels and Log-Level Names*

Number	Log Level
2	Trace
3	Debug
4	Info
5	Warn
6	Error

Number	Log Level
7	Fatal
8	Off

To retrieve an individual logger, enter this command:

```
logger = esp_getLogLevel("DF.ESP")
```

The data looks like this:

```
levelname=info
name=esp.http
level=4}
```

To retrieve all the loggers, enter this command:

```
logger = esp_getLogLevel()
```

The data looks like this:

```
1=
    levelname=error
    name=DF.ESP
    level=6
2=
    levelname=info
    name=DF.ESP.AUTH
    level=4
3=
    levelname=info
    name=common.http
    level=4
...
```

Directly Injecting CSV Data

Use `esp_injectCsv (data,options)` function with the Lua connector to inject CSV data directly without converting it to Lua objects.

Table 8.14 *Parameters of the esp_injectCsv Function*

Parameter	Description
`data`	Specifies a reference to the CSV data.
`options`	Specifies a Lua object with the following fields: ■ `opcode` specifies the opcode to use when the data does not include opcodes.

Parameter	Description
	■ `flags` specifies the flag to use when the data does not include flags. ■ `quiesce` specifies an optional Boolean variable that determines whether the project is quiesced after injection. The default value is **`false`**.

The following code injects broker data in CSV format:

```
<property name="code"><![CDATA[
   local    data = [[
        1012112,Joe,ESP & SAS,SAS Campus Drive Cary NC
27513,919-123-4567
        1012223,Sally,ESP & SAS,SAS Campus Drive Cary NC
27513,919-123-4567
        1012445,Curt,ESP & SAS,SAS Campus Drive Cary NC
27513,919-123-4567
        1012334,Lisa,ESP & SAS,SAS Campus Drive Cary NC
27513,919-123-4567
        1012001,John,ESP & SAS,SAS Campus Drive Cary NC
27513,919-123-4567
   ]]

    function publish()
        esp_injectCsv(data,{opcode="i",flags="n",quiesce=true})
        return true
    end
]]></property>
```

General Utility Functions

Miscellaneous Functions

Table 8.15 *Miscellaneous Functions*

Function	Description
`esp_isDone()`	Calls into the ESP server to check whether the connector has been stopped (for example, when the server is exiting) and returns a Boolean value. When the function returns **`true`** it indicates that the connector has been stopped in the ESP server. The Lua code must break out of its run loop.

Function	Description
`esp_sleep(`*`milliseconds`*`)`	Specifies that the ESP server sleep the specified number of *milliseconds* before returning to the Lua code. This enables you to control how events are published. When the function returns **`false`**, it indicates that the connector has been stopped.The Lua code must break out of its run loop.
`esp_getUTCOffset(`*`time`*`)`	Gets the offset of the system locale, in seconds, from Coordinated Universal Time (UTC) for the specified *time*. The *time* parameter is optional: if it is not specified, then the function uses the current time. When you want to pass in a time, you can create it using `os.time()` as follows: `local offset = esp_getUTCOffset(os.time{year=2024, month=7,day=28})`
`esp_getUTCOffsetHours(`*`time`*`)`	Gets the offset of the system locale, in hours, from Coordinated Universal Time (UTC) for the specified *time*. The *time* parameter is optional: if it is not specified, then the function uses the current time. When you want to pass in a time, you can create it using `os.time()` as follows: `local offset = esp_getUTCOffsetHours(os.time{year=2024,month=7,day=28})`
`esp_inlist(`*`item`*`,`*`strings`*`)`	Looks for *item* in the *strings* array. Example: `e.code = inlist("cary", {"raleigh","durham","chapel hill"})`

`esp_toString(`*`object`*`)`	Converts a Lua *object* into a readable string.
	`function create(data,context)`
	`  -- Write the contents of the input event`
	`  print(esp_toString(data))`
	`  ...`
	`end`

Function	Description

`esp_getServerProperty(name)`	Retrieve the value of a server level property from the ESP server. These are values specified on the server command line, in esp-properties.yml, or in the environment. For example, if you wanted to send a REST request to get the ESP project and schema from the server in which you are running:
	`local host = esp_getServerProperty("HOST")`
	`local port = esp_getServerProperty("http")`
	`local url = "http://"..host..":"..port.."/eventStreamProces`
	
	`local  request = {}`
	
	`request["url"] = url`
	`request["method"] = "GET"`
	`request["headers"] = {accept="application/json"}`
	`request["tolua"] = true`
	`local  response = esp_sendHttp(request)`
	`print(esp_toString(response))`

`esp_getProjectProperty(name, [project])`	Retrieve the value of a project level property. The project parameter is optional. If you do not specify a project name, and there is a single project in the server, that project is used. These values are defined in the project XML:
	`<project>`
	`  <properties>`
	`  <property name="property_1">value 1</property>`
	`  <property name="property_2">value 2</property>`
	`  </properties>`
	`  ...`
	`</project>`

Event Context Functions

The following functions can be useful when Lua code is run in the context of an ESP event. You can use the functions to get the opcode and flag values of that event.

Table 8.16 *Event Context Functions*

Function	Description
`esp_getEventOpcode()`	Returns the opcode of the current event. The returned value is one of the following:

Function	Description
	■ **I** - Insert ■ **U** - Update ■ **D** - Delete ■ **P** - Upsert ■ **SD** - Safe Delete ■ **UB** - Update Block ■ **N** - No op
`esp_getEventFlags()`	Returns the event flags for the current event. The returned value is one of the following: ■ **N** - Normal ■ **P** - Partial Update

Date and Time Functions

Table 8.17 *Date and Time Functions*

Function	Description
`esp_dateMath(`*`date`*`,`*`unit`*`,`*`value`*`)`	Performs calendar math on the specified *date* and return the result as an epoch date. The *value* of *unit* is added to the base date value. Valid unit values are: **second**,**seconds**,**minute**, **minutes**,**hour**,**hours**,**day**, and **days**. Example: `e.seconds = esp_dateMath(os.time( ),"days",-3)` This instance subtracts three days from the current date.
`esp_formatDate(`*`date`*`,`*`format`*`)`	Formats *date* with the specified *format*, which can be any date format

Function	Description
	that is supported by the UNIX `strftime` function. When *format* is not specified, the default system format is used. Example: `e.datestring = esp_formatDate(os.time())`
`esp_formatTime(time,format)`	Formats *time* with the specified *format*, which can be any time format that is supported by the UNIX `strftime` function. When *format* is not specified, the default system format is used. Example: `e.timestring = esp_formatTime(os.time())`
`esp_getSystemMicro()`	Return the number of microseconds since Jan 1, 1970.
`esp_getSystemMilli()`	Return the number of milliseconds since Jan 1, 1970.
`esp_parseDate(datestring,format)`	Parses *datestring* according to the specified *format* and return the epoch date value. *Format* can be any date format that is supported by the UNIX `strftime` function. Example: `e.date = esp_parseDate("2022-04-21","%Y-%m-%d")`
`esp_parseTime(timestring,format)`	Parses *timestring* according to the

Function	Description
	specified *format* and return the epoch time value. *Format* can be any date format that is supported by the UNIX `strftime` function. Example: `e.time = esp_parseTime("2022-04-21T04:09:40","%Y-%m-%dT%H:%M:%S")`
`esp_timeMath(time,unit,value)`	Performs calendar math on the specified *time* and return the result as an epoch time. The *value* of *unit* is added to the base time value. Valid unit values are: **`second`**,**`seconds`**,**`minute`**, **`minutes`**,**`hour`**,**`hours`**,**`day`**, and **`days`**. Example: `e.seconds = esp_timeMath(os.time(),"hours",6)` This instance adds six hours to current time.

String Function

Lua support is limited for operations on strings that contain multibyte characters. The ESP server provides additional support through the `esp_strings` function.

Table 8.18 *String Function*

Function	Description
`esp_strings({func="function",value=value})`	Provide the input value and the function that you wa on it. The ESP server performs the specified operatio string value. Supported functions are as follows: ■ `to_lower` converts the string to lowercase

Function	Description

	■ `to_upper` converts the string to uppercase
	The return value is a Lua table that contains the follo
	■ `value` is the value of the string after the operation an error occurs)
	■ `error` indicates the cause of the problem when on
	Example:
	`local utf8_string = "zaÅ½Ã³Å‚Ä‡ gÄ™Å›lÄ… jaÅºÅ„"`
	`local e = {}`
	`e.id = 1`
	`e.utfstr = esp_strings({func="to_upper",value=utf8_s`
	`...`

Working with Lua Properties

Overview

You can use Lua properties to communicate across Lua entities within a SAS Event Stream Processing project. Lua properties are name-value pairs that are stored in the ESP server. These values are stored in Lua *contexts*. Ordinarily, you do not need to specify a context when you access properties because a default context is automatically created by the ESP server. However, you can create multiple contexts whenever you think it could facilitate more efficient access to Lua properties.

The following code sets the value of `myproperty` in two different contexts:

```
esp_setProperty({name="myproperty",value="one"})
esp_setProperty({name="myproperty",value="one",context="newcontext"})
```

The first call sets `myproperty` in the default Lua context. The second call sets the property in a context named `newcontext`. In both cases, the call sets the value of the property to the hardcoded string **one**. To instead set the value of the property by referencing the value of a variable, omit the quotation marks.

Any Lua entity can read property values.

The ESP server must lock the properties that are associated with the Lua object, but only a READ lock is required to retrieve properties. Properties are locked with a WRITE lock whenever the Lua object that owns the properties changes them. Locking increases the efficiency of working with properties.

The value of a Lua property is a Lua object (string, number, table, and so on) that is stored in the ESP server.

Setting Properties

You can set Lua properties from any Lua code within a project. Use the `esp_setProperty` function to set a Lua property.

Table 8.19 *esp_setProperty Function*

Syntax	Description
`esp_setProperty({name=`*`name`*`,value=`*`value`*`[,field=`*`field`*`][,context=`*`context`*`][,persist=true | false]})`	For `name`, specify the name of the Lua property. For `value`, specify the value of the Lua property (any Lua object). Use the `field` parameter to set a field in the Lua property. Use the `context` parameter to specify a different Lua context in which to set the property. To persist properties, set `persist=true`. Persistent properties are saved to the file system when they are set. The ESP server uses these saved values to initialize the property values upon start-up.

Use dot notation to specify fields within a Lua property. For example, suppose that the following employee information is stored as a Lua property:

```
local   employees = {}

employees.Jose =
{department="Marketing",position="Marketer",info={vacation_days=30,years_of_service=8}}
employees.Sally = {department="Research and
Development",position="Developer",info={vacation_days=30,years_of_service=10}}
employees.Miko = {department="Research and
Development",position="Tester",info={vacation_days=10,years_of_service=3}}
employees.Helen = {department="Support",position="Customer Support
Engineer",info={vacation_days=24,years_of_service=30}}

esp_setProperty({name="employee_data",value=employees})
```

The following Lua code sets vacation days for Jose:

```
esp_setProperty({name="employee_data",value=25,field="Jose.info.vacation_days"})
```

The following Lua window uses a heartbeat to set a Lua property:

```
<window-lua name="lua" events="create" heartbeat="hb">
  <schema>
```

```
    ...
  </schema>
    <code><![CDATA[

        function create(data,context)
            ...
        end

        function hb()
            local   info = {}
            info.foo = "bar"
            info.num = 100
            esp_setProperty({name="info",value=info})
        end

    ]]></code>
</window-lua>
```

The `esp_setProperty` function returns either **`true`** when the property is successfully set or **`false,err`** when the server encounters an error in setting the property. Usually, the call looks like this in order to catch errors:

```
local   code,err = esp_setProperty({name="product",value="Esp"})

if (code == false)
then
    print("cannot set property: "..tostring(err))
end
```

Getting Properties

Any Lua entity can use the `esp_getProperty` function to retrieve a Lua property from the server.

Table 8.20 *esp_getProperty Function*

Syntax	Description
`esp_getProperty({name=`*`name`*`, [field=`*`field`*`] [,context=`*`context`*`]})`	For `name`, specify the name of the property to retrieve. Use the `field` parameter to specify a field to retrieve. Use the `context` parameter to specify a different Lua context from which to retrieve the property.

The function returns the value of the Lua property or **`nil`** when the property is not found. When an error is encountered (for example, when the ESP server cannot find the entity), **`nil`** is returned as a second parameter:

```
local value,err = esp_getProperty({name="foo"})
if err ~= nil then
    ...
end
```

Assume that you wanted to retrieve the entire Lua property of the employee information that is provided in the previous topic. Also assume that the Lua entity is named `test`. Consider the following code:

```
local   data,err = esp_getProperty({name="employee_data"})
print(esp_toString(data))
```

The following output is generated:

```
Helen=
        info=
            years_of_service=30.0
            vacation_days=24.0
        department=Support
        position=Customer Support Engineer
    Jose=
        info=
            years_of_service=8.0
            vacation_days=30.0
        department=Marketing
        position=Marketer
    Miko=
        info=
            years_of_service=3.0
            vacation_days=10.0
        department=Research and Development
        position=Tester
    Sally=
        info=
            years_of_service=10.0
            vacation_days=30.0
        department=Research and Development
        position=Developer
```

Suppose that you wanted the `info` field only for Sally. Consider the following code:

```
local   data,err =
esp_getProperty({name="employee_data",field="Sally.info"})
```

The following output is generated:

```
years_of_service=10.0
vacation_days=30.0
```

Clearing Properties

The `esp_clearProperty` removes the property from the Lua context.

Table 8.21 *esp_clearProperty Function*

Syntax	Description
`esp_clearProperty({name=`*`name`* `[,field=`*`field`*`]`	Use `name` to specify the name of the property to clear. Use the `field` parameter to specify a field of the property to clear. Use `context` to specify a different Lua context in which to clear the

Syntax	Description
[,context=*context*][,persist=true | false]})	property. Use `persist` to specify whether to remove the property value from disk.

The `esp_clearProperty` function returns either **`true`** when the property is successfully cleared, or **`false,err`** when the ESP server encountered an error when attempting to clear the property. Usually, the call looks like this in order to catch any errors:

```
local   code,err = esp_clearProperty({name="product"})

if (code == false)
then
    print("cannot clear property: "..tostring(err))
end
```

Property Listeners

The `esp_addPropertyListener` function enables you to register for changes to properties so that you do not need to continually call `esp_getProperty()`.

Table 8.22 *esp_addPropertyListener Function*

Syntax	Description
esp_addPropertyListener({name="*property*", change="*function*", [clear="*function*"], [context=*context*], [init="true| false"]})	Use the `name` parameter to fetch the name of the *property* to which you want to register. You can specify `*` to fetch all properties. Use the `change` parameter to specify the name of a Lua function that is called when an event changes. Specify the *function* as follows: `function myfunc(property)` Where *property* references a Lua object with the following fields: ■ *name* specifies the name of the relevant property ■ *value* specifies the value of the property ■ *field* points to a field in the property for which you want to receive changes, if applicable

Syntax	Description
	■ *context* specifies the name of a different Lua context in which the property exists Use the `clear` parameter to specify the name of a Lua function to be called when the event is cleared. The function signature is the same as the `change` function except that the `value` field is absent. Use the `context` parameter to specify a different Lua context of the property. Use the `init` to indicate whether you want an initial download of property values. When set to **`true`**, the callback is called for the requested properties. The default value is **`false`**.

Be mindful of the consequences of setting a different context for a property listener. For example, consider the following code.

```
esp_addPropertyListener( {name="ITEMS_RAF234_1_1",change="setFilterCrit
eria",context="FILTER"} )
```

In this code, a listener only gets called when `esp_setProperty` specifies the *FILTER* context.

```
esp_setProperty( {name="ITEMS_RAF234_1_1",value=data.FILTER_CONDITION,c
ontext="FILTER"} )
```

The following code notifies only a listener that is registered in the default context

```
esp_setProperty( {name="ITEMS_RAF234_1_1",value=data.FILTER_CONDITION}
).
```

In this case, the `setFilterCriteria` change function specified in the property listener does not get called.

Suppose that you have a Lua snippet that reads a token from a file every five minutes:

```
<lua-snippet name="access_token">
    <lua-thread func="readToken" interval="5 minutes"/>
    <code><![CDATA[

        function readToken()
            local   f = io.open("token.txt","r")
            local   token = f:read("*all")
            f:close()
            esp_setProperty({name="token",value=token})
        end
```

```
    ]]></code>
</lua-snippet>
```

The following Lua connector uses that token to send HTTP requests:

```
<connector class="lua" name="pub">
  <properties>
    <property name="type">pub</property>
    <property name="code"><![CDATA[

        local   token = nil

        function setToken(property)
            token = property.value
        end


esp_addPropertyListener({name="token",change="setToken",init=true})

        function publish()

            -- use token to make REST request, parse response, and
inject events

            local   request = {}

            request["url"] = "http://somedata.org"
            request["method"] = "GET"
            request["headers"] = {Authorization="Bearer "..token}

            local   response = esp_sendHttp(request)
            ...
        end

    ]]></property>
  </properties>
</connector>
```

Whenever a Lua snippet reads the token from a file and sets the property, the `setToken` function is called and a new value is saved to be used in the HTTP request. Whenever a specific field is set in a property, that field is delivered by the ESP server.

Assume you have written the following handler function:

```
function handler(property,value,field)
    print("employee data changed, field "..property.field)
    print(esp_toString(property.value))
end
```

You now can change the property (make Miko a manager and give him more vacation).

```
...
local   markdata = esp_getProperty({name="test",field="Miko"})
markdata.info.position = "Manager"
esp_setProperty({name="employee_data",value=markdata,field="Miko"})
...
```

When you call the handler function, the following output appears in the console:

```
employee data changed, field Miko

info=
    position=Manager
    years_of_service=3.0
    vacation_days=30.0
department=Research and Development
position=Tester
```

Persistent Properties

Persistent properties are saved to the file system when they are set. The ESP server uses these saved values to initialize the property values upon start-up.

To enable persistent properties on a Lua window or a Lua snippet, set the `persist` property of the `esp_setProperty` or the `esp_clearProperty` function:

```
esp_setProperty({name="myproperty",value="foo",persist=true})
esp_clearProperty({name="myproperty",persist=true})
```

The properties are stored in a directory that is derived from the project's Lua root directory. The Lua root directory can be specified using any of the following methods, in order of precedence:

- An attribute in the project root element:

  ```
  <project
  name="myproject" threads="4" luaroot="/mnt/data/lua_root">
  ```

- A server command-line option: `$ dfesp_xml_server ... -luaroot /mnt/data/lua_root`
- An entry in the esp-properties.yml file in the server section:`server: luaroot: /mnt/data/lua_root`

Properties are stored in a hierarchical manner under the `properties` subdirectory of the specified Lua root directory:

```
properties/
  context/
      property1
      property2
      property...
      propertyN
```

Each property is stored in its own file by its property name.

For example, suppose that you have a Lua window that generates device statistics and sets the properties `last_device` and `info`:

```
<window-lua name="lua" events="create">
  <schema>
    <fields>
        <field name="device_id" type="string" key="true"/>
        <field name="description" type="string"/>
```

```
            <field name="value" type="int32"/>
            <field name="unit" type="string"/>
        </fields>
    </schema>
    <code><![CDATA[

        function create(data)

            esp_setProperty({name="last_device",value=data,persist=true})

            local   info = {}
            info.last_device = data.device_id
            info.timestamp = string.format("%s",os.date())

            esp_setProperty({name="info",value=info,persist=true})

            return data
        end

]]></code>
</window-lua>
```

The following structure exists under the Lua root:

```
properties/
    default/
      info
      last_device
```

These properties are stored as JSON objects:

```
{"description":"Measures
height","device_id":"2","unit":"feet","value":7}
```

When the ESP server starts, it looks for any property files that exist under the Lua root directory. It reads and loads any of these property files into the appropriate Lua context so that they are available upon server start-up.

Calling Functions from Lua Snippets

You can use the `esp_call` function to call functions that are defined within Lua snippets.

```
esp_call({name="snippet_name",func="function_to_call"},parameter,
[parameter,…])
```

For the first parameter of the `esp_call` function, specify a Lua table that contains the following fields:

- `name` - identifies the name of the Lua snippet to reference.
- `func` - specifies the name of the function within the snippet that you want to call.

The remaining parameters are sent to the specified function as data.

The return value is a Lua table that contains the following fields:

- `code` - a Boolean that indicates whether the function was executed successfully.
- `results` - a Lua table (array) of the return values from the function (if successful).
- `error` - an error description (if the call failed).

Consider the following snippet:

```
<lua-snippet name="math">
  <code><![CDATA[

        function divide(dividend,divisor)

            if (divisor == 0)
            then
                error("division by 0,
"..tostring(dividend).."/"..tostring(divisor))
            end

            return dividend / divisor
        end

        ]]></code>
</lua-snippet>
```

You could call the `divide` function as follows:

```
function create(data)

    local   e = {}
    local   dividend = data["dividend"]
    local   divisor = data["divisor"]

    local   response =
esp_call({name="math",func="divide"},dividend,divisor)

    if (response.code == true)
    then
        e["quotient"] = response.results[1]
    else
        e["error"] = response.error
    end

    return(e)
end
```

If there is a division by 0, then the `code` field of the response is **`false`** and the `error` field of the response indicates the error. Otherwise, the `code` field is **`true`** and the `results` field contains a single entry that provides the quotient.

Managing Publisher Adapters

Overview

You can use a set of Lua functions to manage publisher adapters in order to stream data into projects. Using these functions can be a viable alternative to using the Adapter Connector for the same purpose. The functions enable the following actions:

- Create an adapter of a supported type:
  - Audio (`audio`)
  - Bacnet (`bacnet`)
  - SAS Cloud Analytic Services (`cas`)
  - Database (`db`)
  - Azure Event Hubs (`eventhubs`)
  - File and Socket (`fs`)
  - Kafka (`kafka`)
  - Kinesis (`kinesis`)
  - IBM WebSphere MQ (`mq`)
  - MQTT (`mqtt`)
  - OPC-UA (`opcua`)
  - PI Web (`piweb`)
  - QuasarDB (`quasardb`)
  - RabbitMQ (`rmq`)
  - Solace (`sol`)
  - Video Capture (`videocap`)
- Start or restart an adapter
- Stop an adapter
- Set adapter parameters
- Remove an adapter

For more information about adapters, see "Overview of Adapters" in *SAS Event Stream Processing: Connectors and Adapters*.

IMPORTANT These functions are available only through Lua snippets.

Using Callbacks

Your Lua code can interact asynchronously with adapters through the following callback functions.

- `adapter_started(adapter)` - called when the adapter has been started
- `adapter_stopped(adapter, status)` - called when the adapter has been stopped. The `status` parameter is the exit status of the adapter.
- `adapter_stdout(adapter,text)` - called when the adapter writes to standard output.
- `adapter_stderr(adapter,text)` - called when the adapter writes to standard error.
- `adapter_error(adapter,errno)` - called if there is a problem starting the adapter. The `errno` parameter is the system error number.

When you define callback functions in your Lua code, they are called at the appropriate time.

Creating an Adapter

Use the following function to create an adapter:

```
local adapter =
esp_adapter_create({class="type",name="name",parms={params})
```

This function takes a single Lua object as a parameter. That object supports the following fields:

- *type* - specifies the type of the adapter (for example, `db`, `fs`, and so on).
- *name* - specifies an optional name for the instance.
- `parms` - specifies a Lua object, `params`, that contain the adapter parameters (the name-value pairs that are passed to the adapter in the `-C` command line parameter).

For example, the following code creates (but does not start) a File and Socket (`fs`) adapter named `myadapter`:

```
local   parms = {
    fstype="csv",
    url="dfESP://@HOSTNAME@:@pubsub@/p/cq/s"
}

adapter = esp_adapter_create({class="fs",name="myadapter",parms=parms})
```

Note: Here, the ESP server resolves `HOSTNAME` to the host on which the adapter is running, and resolves `pubsub` to the publish/subscribe port used by the ESP server.

The return value of this function is a Lua object that is used as a parameter to subsequent adapter function calls.

When an error occurs, the code field of the response is set to **-1** and an error message is sent:

```
message=Adapter type myadapter is not supported
code=-1
```

Starting an Adapter

Use the following function to start an adapter:

```
local response = esp_adapter_start(adapter)
```

The *adapter* parameter is the return value from the `esp_adapter_create` function.

The response is a Lua object that contains a code and optionally a message field. If the adapter is not currently running, the server attempts to start it and returns a code of `0`. Otherwise, you get an error code of `-1`.

If you have defined the `adapter_started` callback function, it is invoked when the process has successfully started.

Stopping an Adapter

Use the following function to stop an adapter:

```
local response = esp_adapter_stop(adapter)
```

The *adapter* parameter is the return value from the `esp_adapter_create` function.

The response is a Lua object that contains a code and optionally a message field. If the adapter is currently running, the server attempts to stop it and return a code of `0`. Otherwise, you get an error code of `-1`.

If you have defined the `adapter_stopped` callback function, it is invoked when the process has successfully stopped.

Restarting an Adapter

Use the following function to restart an adapter:

```
local response = esp_adapter_restart(adapter)
```

The *adapter* parameter is the return value from the `esp_adapter_create` function. The adapter does not need to be currently running in order to perform a restart. If the adapter is running, the ESP server attempts to stop and then start it. Otherwise, the ESP server attempt only to start the adapter. The appropriate callback functions are invoked, depending on the state of the adapter.

The response is a Lua object that contains a code and optionally a message field. If the adapter is successfully restarted, you get a code of `0`. Otherwise, you get an error code of `-1` with an error message.

Setting Adapter Parameters

You can set adapter parameters at any time. However, parameter values do not take effect until the adapter is started or restarted.

Use the following function to set adapter parameters:

```
local response = esp_adapter_set(adapter,{parm1="a",parm2="b",...})
```

The response is a Lua object that contains a code and optionally a message field. If the adapter parameters are successfully set, you get a code of `0`. Otherwise, you get an error code of `-1` with an error message.

Removing an Adapter

In order to remove an adapter, it cannot be running.

Use the following function to remove an adapter:

```
local response = esp_adapter_remove(adapter)
```

The *adapter* parameter is the return value from the `esp_adapter_create` function.

The response is a Lua object that contains a code and optionally a message field. If the adapter is successfully removed, you get a code of `0`. Otherwise, you get an error code of `-1` with an error message.

Accessing Relational Databases from Lua Components

Overview

You can access relational databases from Lua components through ODBC drivers.

When using an ODBC driver, you establish connectivity to a database through a system data source name (DSN). A DSN and the associated database user credentials are required configuration parameters. The general format of a DSN is as follows:

```
DSN=data_source[;attribute=value[;attribute=value]...]
```

You must specify a *data_source*, which names the desired wire protocol to use for database connectivity. For the data source that you specify, you enumerate one or more relevant attributes. Most wire protocols require that you specify a *userid* and *password* in order to access the database. Many wire protocols also require that you specify a *hostname* that identifies the location of the database.

Each attribute that you specify for the *data_source* overrides the value set in the odbc.ini file that is supplied with SAS Event Stream Processing.

For more information, see "Using DataDirect ODBC Drivers" in *SAS Event Stream Processing: Connectors and Adapters*.

Database connections are stored in Lua contexts. In most cases, you do not need to specify a context when you create a connection because a default context is created by the ESP server.

IMPORTANT Lua component access to the following relational databases is supported.

- Db2
- Greenplum
- MySQL
- Oracle
- PostgreSQL

Use unsupported databases at your own discretion.

Connecting to a Database

Use the `esp_sqlConnect` function to connect to a database. For example:

```
local code,err = esp_sqlConnect(options)
```

In the function, **`options`** is a Lua object that contains the following fields:

- `name` is the name of the SQL connection.
- `connstr` is the DSN, which is described here.
- `pwdencrypted` is an optional Boolean value that specifies whether the password is encrypted.
- `context` is an optional value that specifies the Lua context in which to create the connection.

The name of the connection is used in subsequent database calls.

The `esp_sqlConnect` function returns one or two values. The first value returned is a Boolean that indicates whether the connection was successful. If the connection failed, then the reason for failure is returned through the second value.

For example, the following code shows how to connect to a PostgreSQL database:

```
local connstr = "DSN=PostgreSQL Wire
Protocol;UID=postgres;PWD=mypass123;servername=myserver;port=4444;datab
ase=espdemo" local code,err =
esp_sqlConnect({name="db",connstr=connstr})
```

When you are finished with the connection, you can use the `esp_sqlDisconnect` function to remove it.

```
local code,err = esp_sqlDisconnect(options)
```

In the function, **`options`** is a Lua object that contains the following fields:

- `name` is the name of the SQL connection to disconnect.
- `context` is an optional value that specifies the Lua context in which the connection exists.

The `esp_sqlDisconnect` function returns one or two values. The first value returned is a Boolean that indicates whether the disconnection was successful. If the disconnection failed, then the reason for failure is returned through the second value.

The following code shows how to disconnect a Postgres connection:

```
local code,err = esp_sqlDisconnect({name="db"})
```

Retrieving Data

There are two ways to retrieve data from a database. The first way obtains all the results from an SQL query by sending an SQL select statement to the ESP server.

```
local resp = esp_sqlSelect(options)
```

In the statement, **`options`** is a Lua object that contains the following fields:

- `conn` is the name of the SQL connection to use.
- `query` is the SQL select statement.
- `context` is an optional value that specifies the Lua context in which the connection exists.

The return value is a Lua table that contains the following fields:

- `code` is a Boolean value that indicates whether the query was successful.
- `data` is the database data.
- `message` is an informative message sent when the query fails.

You can use `esp_sqlSelect` with moderately sized queries. For example, suppose millions of stock trades are stored in a trades table. The following code shows how to retrieve extremely large trades (of more than a million shares):

```
local resp = esp_sqlSelect({conn="db",query="select * from trades
where quant > 1000000"})

print(esp_toString(resp))

code=true
```

```
data=
    1=
        broker=101667
        buyer=0
        msecs=724
        symbol=XEL
        buysellflg=1
        price=21.7
        seller=8123541
        currency=87236
        quant=1286200
        id=18564845
        time=1280928728
        venue=55555
    2=
        broker=101667
        buyer=1012223
        msecs=418
        symbol=C
        buysellflg=0
        price=4.15
        seller=8382739
        currency=87236
        quant=2063400
        id=2669875
        time=1280928609
        venue=55555
```

However, selecting all of the trades in one pass would consume too much memory. When you need to retrieve large amounts of data from the database, it is recommended to retrieve data in increments. Use the following functions to open a query and retrieve the data in increments:

- `esp_sqlOpen` - opens a query.
- `esp_sqlNext` - retrieves a specified number of rows from the result.
- `esp_sqlClose` - closes a query.

Use the `esp_sqlOpen` function as follows:

```
local handle = esp_sqlOpen(options)
```

In the function, **`options`** is a Lua object that contains the following fields:

- `conn` is the name of the SQL connection to the ESP server.
- `query` is the SQL select statement.
- `context` is an optional value that specifies the Lua context in which the connection exists.

The return value is a Lua table that contains the following fields:

- `code` is a Boolean value that indicates whether the query was successful.
- `message` is an informative message that is sent when the query fails.

After you successfully open a query, you can retrieve the data with the `esp_sqlNext` function.

```
local data,count = esp_sqlNext(handle,numrows)
```

- `handle` is the value that is returned from `esp_sqlOpen`.
- `numrows` is the number of rows to return.

If you do not specify a `numrows` parameter, then the remainder of the rows in the result are returned.

The function returns two values. The first value is the data from the query result, or **`nil`** when an error occurs. The second value is the number of records returned, or **`nil`** when an error occurs or when the data has been exhausted. The data is exhausted whenever the function returns a count of 0. When that happens, the server closes the query, so you do not need to call `esp_sqlClose`.

When you are finished with the query, use the `esp_sqlClose` function to remove the query from the server.

```
local code,err = esp_sqlClose(handle)
```

The function returns two values. The first value is a Boolean that indicates whether the close was successful. The second value contains an error message that provides information about the cause of the failure.

As previously mentioned, when you have retrieved all the rows you do not need to close the query because the ESP server does it for you.

Inserting Data

Use the `esp_sqlInsert` function to insert records into the database.

```
local resp = esp_sqlInsert(options)
```

In the function, **`options`** is a Lua table that contains the following fields:

- `conn` is the name of the SQL connection to use.
- `table` is the database table in which to insert data.
- `data` contains one or more Lua objects that contain the records to insert.
- `context` is an optional value that specifies the Lua context in which the connection exists.

If you are processing stock trades, then you could use a Lua subscriber connector on the Source window to grab incoming trades and write them to a database:

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

```
            <field name="seller" type="int32"/>
            <field name="buysellflg" type="int32"/>
        </fields>
    </schema>
    <connectors>
        <connector class="lua" name="sub">
            <properties>
                <property name="type">sub</property>
                <property name="code"><![CDATA[
                    function subscribe(data)
                        local   response =
esp_sqlInsert({conn="db",table="trades",data=data})
                        if (response.code == false)
                        then
                            print(esp_toString(response))
                        end
                    end
                ]]></property>
            </properties>
        </connector>
    </connectors>
</window-source>
```

The return value from `esp_sqlInsert` is a Lua object that contains the following fields:

- `code` is the status of the insert: **`true`** for success and **`false`** for failure.
- `message` contains information that is related to the failure when one occurs.
- `object` is the data for the failed record when a failure occurs.

Updating Data

Use the `esp_sqlUpdate` function to update the database.

```
local resp = esp_sqlUpdate(options)
```

In the function, **`options`** is a Lua table that contains the update properties.

There are two ways that you can update the database through Lua components. The first way is to send only the data that contains the fields to update. When you use this approach, the following fields are accepted in the `options` parameter:

- `conn` is the name of the SQL connection to use.
- `table` is the database table to update.
- `data` contains one or more Lua objects that contain the update.
- `upsert` is an optional Boolean value that attempts an insert of the record when one does not exist. The default value is **`false`**.
- `context` is an optional value that specifies the Lua context in which the connection exists.

The server derives the database key from the data, so the data must contain at least all key fields in each object. This update method can accept multiple Lua objects in the function call. Each object generates an individual update request to the database.

For example, suppose you are processing stock trades and want to save the minimum and maximum prices for each symbol. Create the following table:

```
create table trades_data
(
    symbol text,
    min_price float,
    max_price float,
    range float,
    primary key (symbol)
);
```

The following code streams trades and keeps the current minimum and maximum price for each symbol:

```
local   resp = esp_sqlUpdate({conn="db",
                              table="trades_data",
                              upsert=true,
                              data={

{symbol="ABC",min_price=100,max_price=120,range=20},

{symbol="XYZ",min_price=10,max_price=20,range=10}
                              }})
```

Note that the `upsert` value is **`true`**. As a result, the server adds the record when it does not exist.

The second way to perform an update is to add a single Lua object with an SQL WHERE clause that identifies the database records to update. In the following code, the data from the request is applied to all records that match the WHERE clause:

```
local   resp = esp_sqlUpdate({conn="db",
                              table="trades_data",
                              data={min_price=0,max_price=0,range=0},
                              where="1=1"})
```

Any key fields in the data are ignored. The `trades_data` table from the previous example is initialized, zeroing previously existing values.

The return value from `esp_sqlUpdate` is a Lua object that contains the following fields:

- `code` is the status of the update: **`true`** for success and **`false`** for failure.
- `message` is information that is related to the failure when one occurs.
- `object` is the data for the failed record when a failure occurs.

Deleting Data

Use the `esp_sqlDelete` function to delete records from a database.

```
local resp = esp_sqlDelete(options)
```

In the function, **`options`** is a Lua table that contains the delete properties.

There are two ways that you can delete records from the database through Lua components. The first way is to send one or more Lua objects that contain key field data of the records to delete. When you use this approach, the `options` parameter accepts the following fields:

- `conn` is the name of the SQL connection to use.
- `table` is the database table from which to delete records.
- `data` contains one or more Lua objects that contain the key fields of the records to delete.
- `context` is an optional value that specifies the Lua context in which the connection exists.

The following code deletes a few records from the `trades_data` table in the update section:

```
local   resp = esp_sqlDelete({conn="db",
                              table="trades_data",
                              data={{symbol="ABC"},{symbol="XYZ"}}})
```

The second way to delete records is to add an SQL WHERE clause that identifies the database records to delete. The following code deletes all the records from `trades_data` that meet a certain condition (`max_price > 10`:

```
local resp =
esp_sqlDelete({conn="db",table="trades_data",where="max_price > 10"})
```

The return value from `esp_sqlDelete` is a Lua object that contains the following fields:

- `code` is the status of the delete: **`true`** for success and **`false`** for failure.
- `message` is information that is related to the failure when one occurs.
- `object` is the data for the failed record when one occurs.

Using Lua Snippets

Lua snippets are self-contained blocks of Lua code. You create Lua snippets at the project level.

Use the `lua-snippet` element to define the snippet. For more information, see "lua-snippet" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Use the `code` element to specify the code for the snippet.

The code can be either inline or in an external file. If it is in an external file you can use the file attribute to point to the file.

```
<code file="mycode.lua" />
```

Consider the following snippet. After it runs initialization and cleanup functions, it opens a file named **`token.txt`** and reads a token from it. Then the snippet sets a property called `token` in the ESP server. This property can be accessed by other Lua entities.

```
<project...>
   <lua-snippets>
    <lua-snippet name="access_token" init="initfunc" destroy="cleanup">
        <code><![CDATA[
            function initfunc()
                print("initializing snippet")
            end

            function cleanup()
                print("cleaning up snippet")
            end

            function load()
                local   f = io.open("token.txt","r")
                local   token = f:read("*all")
                f:close()
                esp_setProperty({name="token",value=token})
            end

            load()

        ]]></code>
    </lua-snippet>
    ...
</project>
```

Consider the following Lua connector. It uses the token set by the Lua snippet to make a REST request:

```
<connector class="lua" name="pub">
  <properties>
    <property name="type">pub</property>
    <property name="interval">5 seconds</property>
    <property name="code"><![CDATA[

        local   token = esp_getProperty({name="token"})

        function publish()
            -- use token to make REST request, parse response, and
inject events
            return true
        end
```

```
    ]]></property>
  </properties>
</connector>
```

Note: The `esp_getProperty` function takes the name of the Lua property.

Lua snippets also enable you to start a thread that calls a specified function at a specified interval. The following code revises the previous snippet so that when the token expires after a period, the snippet gets a new token every five minutes and sets the property.

```
<project ...>
  <lua-snippets>
    <lua-snippet name="largetrades">
        <lua-thread func="load" interval="5 minutes" />
        <code><![CDATA[

            function load()
                local   f = io.open("largetrade.txt","r")
                local   contents = f:read("*all")
                f:close()
                esp_setProperty({name="token",value=token})
                return true
            end

            load()

        ]]></code>
    </lua-snippet>
  </lua-snippets>
  ...
</project>
```

The return code from the function is a Boolean value that tells the server whether to keep the thread running. If you return **`true`**, the thread continues running. If you return **`false`**, the thread terminates.

Referencing the revised snippet, the Lua connector can grab the token each time it publishes to make sure it has a fresh one to work with its request.

```
<connector class="lua" name="pub">
  <properties>
    <property name="type">pub</property>
    <property name="code"><![CDATA[

        function publish()
            local   token = esp_getProperty({name="token"})
            -- use token to make REST request, parse response, and
inject events
            return true
        end

    ]]></property>
  </properties>
</connector>
```

The code within a Lua snippet is loaded in the order in which snippets are declared in the XML. When a snippet is declared before another requires variables set in a subsequent snippet, those variables are not available.

Consider the following code:

```
<lua-snippets>
    <lua-snippet name="thisSnippet">
        <code><![CDATA[

            esp_setProperty({name="theproperty",value="hello, world"})

        ]]></code>
    </lua-snippet>
    <lua-snippet name="thatSnippet">
        <code><![CDATA[

            local   value = esp_getProperty({name="theproperty"})
            print("VALUE IS: "..tostring(value))

        ]]></code>
    </lua-snippet>
</lua-snippets>
```

When these snippets are included as specified, the code returns the following output in the ESP log:

```
VALUE IS: hello, world
```

Suppose that the snippets are transposed.

```
<lua-snippets>
    <lua-snippet name="thatSnippet">
        <code><![CDATA[

            local   value = esp_getProperty({name="theproperty"})
            print("VALUE IS: "..tostring(value))

        ]]></code>
    </lua-snippet>
    <lua-snippet name="thisSnippet">
        <code><![CDATA[

            esp_setProperty({name="theproperty",value="hello, world"})

        ]]></code>
    </lua-snippet>
</lua-snippets>
```

The following output is returned to the ESP log because the code for `thatSnippet` is run before the code for `thisSnippet`.

```
VALUE IS: nil
```

For information about the start-up sequencing of Lua entities, see “Initialization of Lua Entities”

Using Lua Modules

Lua modules are self-contained blocks of Lua code that exist at the project level. The code in a module exists only to be copied into other Lua components. Lua modules enable you to write code a single time for reuse in any number of other Lua components.

```
<project name="name"...>
  <lua-modules>
    <lua-module name="mod_1">
      <code><![CDATA[
       -- Lua code
       ]]></code>
    </lua-module>
    <lua-module name="mod_2">
      <code><![CDATA[
       -- Lua code
       ]]></code>
    </lua-module>
  </lua-modules>
  <contqueries>
    <contquery name="cq1">
    ...
    </contquery>
  </contqueries>
</project>
```

Specify the `require` attribute in other Lua components to reference Lua modules.

```
<lua-module name="mod_1" require="mod_2"> <!-- 1 -->
<window-lua name="name" require="mod_1,mod_2">
<lua-snippet name="name"  require="mod_1,mod_2">
<window-filter name="name" require="mod_1,mod_2">
<window-pattern name="name">
     ...
     <code require="mod_1,mod_2"> <!-- 2 -->
```

1 A Lua module can call another Lua module.

2 Use the `require` attribute in the `code` element of a Pattern window that uses Lua code.

The Lua connector can bring in Lua modules with a `require` property:

```
<connectors>
   <connector class="lua" name="pub">
     <properties>
       <property name="type">pub</property>
       <property name="interval">500 milliseconds</property>
       <property name="init">init</property>
       <property name="require">mod_1,mod_2</property>
     </properties>
```

```
    </connector>
  </connectors>
```

Suppose that you wanted a class to encapsulate the ESP server logging and make it available to any Lua entity:

```
<lua-module name="logger">
    <code><![CDATA[
        Logger = {
            _context = nil
        }

        function Logger:new(context)
            o = {}
            setmetatable(o,self)
            self.__index = self
            self._context = context or "mylogger"
            return o
        end

        function Logger:error(msg,line)
            esp_logMessage(self._context,msg,"error",line)
        end

        function Logger:warn(msg,line)
            esp_logMessage(self._context,msg,"warn",line)
        end

        function Logger:fatal(msg,line)
            esp_logMessage(self._context,msg,"fatal",line)
        end

        function Logger:info(msg,line)
            esp_logMessage(self._context,msg,"info",line)
        end

        function Logger:debug(msg,line)
            esp_logMessage(self._context,msg,"debug",line)
        end

        function Logger:trace(msg,line)
            esp_logMessage(self._context,msg,"trace",line)
        end


    ]]></code>
</lua-module>
```

Suppose that you have a Lua window that divides numbers coming into an event. If the divisor is 0, you want to log an ERROR message to the server with the Logger class from this module. In the code that follows, the `require` attribute brings in the module that contains the Logger class.

```
<window-lua name="lua" events="create" require="logger">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="dividend" type="int32"/>
```

```
      <field name="divisor" type="int32"/>
      <field name="quotient" type="double"/>
    </fields>
  </schema>
  <copy>id,dividend,divisor</copy>
  <use>dividend,divisor</use>
  <code><![CDATA[
    function create(data)

        local   e = {}
        local   dividend = data["dividend"]
        local   divisor = data["divisor"]

        if (divisor == 0)
        then
            local   logger = Logger:new("modules.example")
            logger:error("division by 0,
"..tostring(dividend)..”/”..tostring(divisor))
            return nil
        end

        e["quotient"] = dividend / divisor

        return(e)
    end

  ]]></code>
</window-lua>
```

The Lua code creates an instance of the class and uses it to log a message to ESP server log. Errors appear in the server log whenever a division by 0 occurs.

```
{
    "level":"ERROR",
    "message":"{6c00732f-5d9d-4203-9c42-731d4e442b30}[XMLServer0001] division by
0, 31\/0",
    "properties":
    {
        "caller":"Lua.cpp:1910",
        "logger":"modules.example",
        "thread":"00000033"
    },
    "timeStamp":"2023-09-13T16:19:04.891000-04:00"
}
```

Handling Exceptions in Lua Code

Lua code can throw exceptions. Exceptions from the `init` function cause the project to fail to load. Exceptions from the `create` function, the `heartbeat` function, and snippets (when threaded) are handled based on the `on-script-exception` attribute of the `project` element. For more information, see “project” in

SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models.

9

Common Python Functionality

User-Provided Code in ESP Projects

SAS Event Stream Processing enables you to extend your projects with your own code (for example, Python, Lua, or other custom components). You and your organization are solely responsible for the behavior and security of any code you decide to use. SAS Event Stream Processing does not review or test user-provided

code for security purposes. Any code or other customer-supplied materials you add to an ESP project remain your responsibility.

CAUTION

Do not run unsafe code. Running unsafe code may expose your environment to data loss, data corruption, security breaches, or system instability. It is your responsibility to ensure that all code you incorporate is secure, tested, and compliant with your organization's policies and applicable law.

Initialization of Python Entities

The ESP Server provides several Python entities as well as communication between them. There are also persistent properties that are saved to disk and read back in on server start-up. It is important to know how all these things are initialized when the server starts.

All Python entities provide an initialization mechanism. The Python window implements this mechanism through an `init` attribute in the model XML.

The Python connector provides an optional `init` parameter that points to an initialization function specified in the Python code.

It is important to use initialization functions when you have Python entities that obtain property values from other Python entities. For example, the following code might produce a `None` value because the other Python entity might not have set the value property when the code is called.

```
myvalue = esp.getProperty(name="value")
```

However, if you place the action of obtaining a property value within an initialization function, you can be sure that the property has been set if the other entity has been initialized:

```
myvalue = None
def init():
    myvalue = esp.getProperty(name="value")
```

When the ESP server starts, Python-related actions are performed in the following sequence:

1. Loads persistent properties for Python snippets.
2. Loads persistent properties for Python windows.
3. Calls the `init` function for any Python snippet that has one.
4. Starts any threaded Python snippets.
5. For any Python window, calls the `init` function if it has one. Also, when the window has a Python splitter, it calls the `init` function on that.

All of the above operations are performed in the order the entities are defined in the XML model.

Using SAS Event Stream Processing Functions with Python

Overview

The ESP server provides functions that you can invoke within Python code that enable the following activities. There are two packages, which are called esp and esp_utils. The esp package contains functions that communicate with the ESP server. The esp_utils package is used for working with ONNX models and images. The following tables organize the functions by packages and activities.

In order to use esp functions, include the following line in your Python code:

```
import esp
```

Table 9.1 *esp Functions Organized by Activity*

Activity	Function
"Working with Events"	`esp.publish` `esp.query`
Working with HTTP requests	`esp.sendHttp`
"ESP Server Log Debugging"	`esp.logMessage`

"Working with Python Properties"	`esp.setProperty`
	`esp.getProperty`
	`esp.clearProperty`

"Calling Functions from Python Snippets"	`esp.call`
"Miscellaneous Functions"	`esp.getServerProperty` `esp.getProjectProperty`

Accessing Relational Databases	`esp.sqlConnect`
	`esp.sqlSelect`
	`esp.sqlOpen`

Activity	Function
	esp.sqlNext
	esp.sqlInsert
	esp.sqlUpdate
	esp.sqlDelete

In order to use esp_utils functions, include the following line in your Python code:

```
import esp_utils
```

Table 9.2 *esp_utils Functions Organized by Activity*

Activity	Function
"Manipulating Images"	esp_utils.image_conversion.sas_wide_image_to_opencv_image esp_utils.image_conversion.opencv_image_to_sas_wide_image esp_utils.image_conversion.blob_image_to_opencv_image esp_utils.image_conversion.opencv_image_to_blob_image
"Working with Tensors"	np_array_to_tensor tensor_to_np_array
"Postprocessing"	bbox_letterbox_to_original xy_letterbox_to_original

Working with Events

Publishing Events

This topic is relevant to the esp package. Use the esp.publish function to publish events to Source windows.

```
local response = esp.publish(window="cq2/storeAlerts",events=events)
```

Table 9.3 *Parameters of the esp.publish Function*

Parameter	Description
window	Specifies the name of the Source window.

Parameter	Description
`events`	Specifies the event data to publish, which is either an array or a single object.
`opts`	Specifies publish options. ■ `blocksize` - specifies the size of the event block to publish. The default value is **1**.

The `esp.publish` function puts an event into a queue to be processed asynchronously. The return value contains a code field and a message field.

Querying Events

This topic is relevant to the esp package. Use the `esp.query` function to query events:

```
patients = esp.query(window="patients")
vitals = esp.query(window="vitals")
```

Table 9.4 *Parameters of the esp.query Function*

Parameter	Description
`window`	Specifies the window from which to receive events.
`output`	Specifies a Python object with the following entries: ■ `fields` - specifies a comma-separated list of fields to include in event delivery. ■ `exclude` - a Boolean value that indicates whether items that are specified in `fields` should be included or excluded. The default value is **`false`**. ■ `encode_binary` - a Boolean value that indicates whether to Base64-encode binary data for event delivery. The default value is **`false`**.
`matches`	Specifies a Python object that contains event fields on which to match. For example: `data = esp.query(window="measurement",matches={"patient":"Patient 1"})`
`sort`	Specifies a Python object that is used to sort events. ■ `fields` - specifies a comma-separated list of fields on which to sort. ■ `dir` - specifies the sort direction, **`ascending`** or **`descending`**. The default is .**`descending`**. ■ `maxevents` - specifies the maximum number of events to return. By default, all events are returned.

Suppose that you had a project that processed stock trades and you wanted to sort the top 10 trades by quantity. The following code uses `esp.query` to obtain events from a window named `trades` and then sorts the output accordingly.

```
events = esp.query(window="p/cq/trades",

                   output={"fields":"symbol,quantity,price"},

                   sort={"fields":"quantity","maxevents":10})
print("events\n" + str(events))
```

Sending HTTP Requests and Receiving Responses

This topic is relevant to the esp package. You can send HTTP requests, and then use the `esp.sendHttp(`*`fields`*`)` function to direct responses back into the Python code of a Python-based window. Format a request object and pass it to the function with the following *fields*:

Table 9.5 *Request Object Fields*

Field	Description
`url`	Specifies the URL of the request.
`method`	Specifies the HTTP method to use (**`GET`**, **`PUT`**, **`POST`**, **`DELETE`**). The default value is **`GET`**.
`headers`	Specifies HTTP request headers to send to the server.
`body`	Specifies the body of the HTTP request for **`PUTs`** and **`POSTs`**.
`topython`	When the HTTP response is in JSON or XML, you can set this field to **`true`** in order for the ESP server to convert the response into a Python object before sending the response back.
`connect-timeout`	Specifies the time-out in number of seconds to wait before successfully establishing a connection with the server. The default value is `0` (no time-out).
`data-timeout`	Specifies the time-out in number of seconds to wait before successfully receiving data from the server. The default value is `0` (no time-out).

Consider the following code:

```
import esp

def create(data,context):
```

```
    url = "http://api.openweathermap.org/data/2.5/group?
id=4487042&APPID=d0b38229a8f66dac4fbce8bf76be97b4&units=imperial"
    response =
esp.sendHttp(url=url,method="GET",headers={"accept":"application/
json"},topython=True)

    print(str(response))
```

When you execute the preceding code within a Python-based window, it should elicit the following response from the weather API:

```
status=200 response={"cnt":1,"list":[{"coord":
{"lon":-78.6386,"lat":35.7721},"sys":
{"country":"US","timezone":-14400,"sunrise":1650450904,"sunset":1650498685},"weath
er":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}],"main":
{"temp":40.91,"feels_like":40.91,"temp_min":35.08,"temp_max":46.42,"pressure":1031
,"humidity":89},"visibility":10000,"wind":{"speed":0,"deg":0},"clouds":
{"all":20},"dt":1650456592,"id":4487042,"name":"Raleigh"}]}
```

Debugging

Overview

You can add diagnostic output directly into Python functions. There are two ways to view this output:

- through the console
- through the ESP server log

Console Debugging

Use the Python `print()` function to print information to the console.

For example, suppose that within a Pattern window you are looking at sequences in stock trade information. The following code dumps the current event along with the context when the function is run. At the end of the function, it dumps the return value:

```
def f3(event,context):
    print("in function e3")
    print(str(event))
    print(str(context))

    code = event["broker"] == "George"
            and event["price"] < context["events"]["john"]["price"]
            and event["price"] < context["events"]["paul"]["price"]
```

```
        print("returning: " + str(code) + " from f3")

        return code
```

Running the code produces output such as this example:

```
in function e3

            {'price':50.0, 'quantity':50, 'broker':'George', 'id':6,
'symbol':'IBM'}

            name':'george'
            window':1,
            events':{'john': { 'price':97.34 'quantity':500 'broker':'John'
'id':0 'symbol':'IBM' }},
                    {'paul': { 'price':57.34 'quantity':500 'broker':'Paul'
'id':3 'symbol':'IBM' }}
            }

        returning: true from f3
```

When you dump the events and context, you can follow the patterns and determine why certain events trigger certain patterns.

ESP Server Log Debugging

This topic is relevant to the esp package. You can use the `esp.logMessage` function to log messages from Python to the ESP server log. The `esp.logMessage` function takes the following parameters:

Table 9.6 *Parameters for the esp.logMessage Function*

Parameter	Description
`logcontext`	Specifies the context object in the ESP server (for example, **`DF.ESP`**). This parameter is required. The ESP server creates the context if it does not exist.
`message`	Specifies the message that you want to log. This parameter is required.
`level`	Specifies the log level to use. If the specified log context is set to that log level or higher, the message is logged. Otherwise, the message is discarded. Possible values of `level` are as follows: ■ **`trace`** ■ **`error`** ■ **`warn`** ■ **`fatal`**

Parameter	Description
	■ **info** ■ **debug** This parameter is optional and defaults to **info**.

line	Specifies the line number of the Lua code from which the message is logged. This value is generated with the following Python call:
	`import inspect`
	`inspect.getframeinfo(inspect.currentframe()).lineno`
	If you do not specify a value for `line`, no Python line number is logged.

Consider the following function that receives an event and generates an event with some minor data modifications. The code logs the incoming event and the generated event if the `mylogger` logging context is set to **debug** or higher:

```
import inspect
import esp

    eventId = 1

function create(data,context)
    global eventId

    esp.logMessage(logcontext="mylogger",message="creating event from"
+
str(data),level="debug",line=inspect.getframeinfo(inspect.currentframe(
)).lineno)

    e = {}

    e["id"] = str(eventId)
    e["new_string"] = "\"" + data["string"] + ""\""
    e["new_number"] = data["number"] * 3
    e["new_array"] = []
    for value in data.array:
        e["new_array"].append(value * 2)

    eventId += 1

    esp.logMessage(logcontext="mylogger",message="created event " +
str(e),level="debug",line=inspect.getframeinfo(inspect.currentframe()).
lineno)

    return(e)
```

The log output might look similar to this example:

```
_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] creating
event from
    id=1
    string=string data
    array=
        1=11
        2=24
        3=55
    number=33
 (Python line 7)
messageFile=./src/dfESPwindow_python.cpp
messageLine=774

_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] created event
    id=1
    new_string="string data"
    new_array=
        1=22
        2=48
        3=110
    new_number=99
 (Python line 21)
messageFile=./src/dfESPwindow_python.cpp
messageLine=774
```

The following code shows server logging for a pattern matching function:

```
def f3(event,context):

        message = "in function e3"
        message = message + "\n" + str(event)
        message = message + "\n" + str(context)

        code = event["broker"] == "George"
                and event["price"] < context["events"]["john"]["price"]
                and event["price"] < context["events"]["paul"]["price"]

        message = message + "\n" + "returning: " + str(code) + " from
f3"

        esp.logMessage(logcontext="mypatternlogger",
                       message=message,
                       level="debug",

line=inspect.getframeinfo(inspect.currentframe()).lineno)

        return code
```

If the `mypatternlogger` logging context is set appropriately, you should see something like this example in the ESP server log:

```
{
      "_timestamp": "2021-12-16T12:36:43.312000-05:00",
      "logger": "mypatternlogger",
      "loggerLevel": "DEBUG",
      "messageContent": "{f3b6119e-e65c-4ad9-a09b-36075fc43339}[XMLServer0001] in
function e3\n\n    symbol=IBM\n    quantity=50\n    id=6\n    broker=George\n
price=50.0\n\n\n    window=1\n    name=george\n    events=\n        paul=
\n            symbol=IBM\n            quantity=500\n            id=3\n
broker=Paul\n            price=57.34\n        john=\n            symbol=IBM
\n            quantity=500\n            id=0\n            broker=John
\n            price=97.34\n\nreturning: true from f3 (Python line 23)",
      "messageFile": "dfESPwindow_pythonbase.cpp",
      "messageLine": "130",
      "thread": "00000034"
}
```


You can get the current log level of loggers in the ESP server with the `esp.getLogLevel()` function. This function returns information about an individual logging context or all contexts if no name is specified.

Both the numeric level and level name are returned in the data.

Table 9.7 *Numeric Levels and Log-Level Names*

Number	Log Level
2	Trace
3	Debug
4	Info
5	Warn
6	Error
7	Fatal
8	Off

To retrieve an individual logger, enter this command:

```
logger = esp.getLogLevel("DF.ESP")
```

The data looks like this:


```
{
    "level": 6,
    "levelname": "error",
    "name": "DF.ESP"
}
```


To retrieve all the loggers, enter this command:

```
logger = esp.getLogLevel()
```

The data looks like this:

```
[
  {
    "level": 6,
    "levelname": "error",
    "name": "DF.ESP"
  },
  {
    "level": 4,
    "levelname": "info",
    "name": "DF.ESP.AUTH"
  },
  {
    "level": 4,
    "levelname": "info",
    "name": "common.http"
  },
  ...

]
```

Working with Python Properties

Overview

The following topics are relevant to the esp package. You can use Python properties to communicate across Python entities within a SAS Event Stream Processing project. Python properties are name-value pairs that are stored in the ESP server. These pairs are stored in Python contexts. Ordinarily, you do not need to specify a context when you access properties because a default context is automatically created by the ESP server. However, you can create multiple contexts whenever you think it could facilitate more efficient access to Python properties.

The following code sets the value of `myproperty` in two different contexts:

```
esp.setProperty(name="myproperty",value="foo")
esp.setProperty(name="myproperty",value="foo",context="newcontext")
```

The first call sets `myproperty` in the default Python context. The second call sets the property in a context named `newcontext`. In both cases, the call sets the value of the property to the hardcoded string **one**. To instead set the value of the property by referencing the value of a variable, omit the double quotation marks.

Any Python entity can read property values.

The ESP server must lock the properties that are associated with the Python object, but only a READ lock is required to retrieve properties. Properties are locked with a WRITE lock whenever the Python object that owns the properties changes them. Locking increases the efficiency of working with properties.

The value of a Python property is a Python object (string, number, table, and so on) that is stored in the ESP server.

Setting Properties

You can set Python properties from any Python code within a project. Use the `esp.setProperty` function to set a Python property.

Table 9.8 *esp.setProperty Function*

Syntax	Description
`esp.setProperty(name=`*`name`*`,value=`*`value`*`[,field=`*`field`*`] [,context=`*`context`*`] [,persist=true | false])`	For `name`, specify the name of the Python property. For `value`, specify the value of the Python property (any Python object). Use the `field` parameter to set a field in the Python property. Use the `context` parameter to specify a different Python context in which to set the property. To persist properties, set `persist=true`. Persistent properties are saved to the file system when they are set. The ESP server uses these saved values to initialize the property values upon start-up.

Use dot notation to specify fields within a property. For example, suppose that the following employee information is stored as a property:

```
employees = {}

employees["Joe"] =
{"department":"Marketing","position":"Marketer","info":
{"vacation_days":30,"years_of_service":8}}
employees["Sally"] = {"department":"Research and
Development","position":"Developer","info":
{"vacation_days":30,"years_of_service":10}}
employees["Mark"]={"department":"Research and
Development","position":"Tester","info":
{"vacation_days":10,"years_of_service":3}}
employees["Helen"]={"department":"Support","position":"Customer
Support Engineer","info":{"vacation_days":24,"years_of_service":30}}

esp.setProperty(name="employee_data",value=employees)
```

The following code sets vacation days for Joe:

```
esp.setProperty(name="employee_data",value=25,field="Joe.info.vacation_
days")
```

The following Python window uses a heartbeat to set a property:

```
<window-python name="python" events="create" heartbeat="hb">
```

```
<schema>
  ...
</schema>
  <code><![CDATA[

      def create(data,context):
          ...

      def hb():
          info = {}
          info["foo"] = "bar"
          info["num"] = 100
          esp.setProperty(name="info",value=info)

  ]]></code>
</window-python>
```

The `esp.setProperty` function returns either **`true`** when the property is successfully set or **`false,err`** when the server encounters an error in setting the property. Usually, the call looks like this example in order to catch errors:

```
code,err = esp.setProperty(name="product",value="Esp")

if (code == False):
    print("cannot set property: "..err)
```

Getting Properties

Any Python entity can use the `esp.getProperty` function to retrieve a Python property from the server.

Table 9.9 *esp.getProperty Function*

Syntax	Description
`esp.getProperty(name=`*`name`*`, [field=`*`field`*`] [,context=`*`context`*`])`	For `name`, specify the name of the property to retrieve. Use the `field` parameter to specify a field to retrieve. Use the `context` parameter to specify a different Python context from which to retrieve the property.

The function returns the value of the property or **`nil`** when the property is not found. When an error is encountered (for example, when the ESP server cannot find the entity), **`nil`** is returned as a second parameter:

```
data = esp.getProperty(name="employee_data")
print(str(data))
```

Suppose that you wanted to retrieve the entire Python property of the employee information that is provided in the previous topic. Also suppose that the Python entity is named `test`.

The following output is generated:

```
{'Helen': {'department': 'Support', 'info': {'vacation_days': 24.0,
'years_of_service': 30.0}, 'position': 'Customer Support Engineer'},
    'Joe': {'department': 'Marketing', 'info': {'vacation_days': 30.0,
'years_of_service': 8.0}, 'position': 'Marketer'},
    'Mark': {'department': 'Research and Development', 'info': {'vacation_days':
10.0, 'years_of_service': 3.0}, 'position': 'Tester'},
    'Sally': {'department': 'Research and Development', 'info': {'vacation_days':
30.0, 'years_of_service': 10.0}, 'position': 'Developer'}}
```

Suppose that you wanted the `info` field only for Sally. Consider the following code:

```
data = {"name": "employee_data", "field": "Sally.in"}
err = esp.getProperty(data)
```

The following output is generated:

```
{'vacation_days': 30.0, 'years_of_service': 10.0}
```

Note: Strings that contain multi-byte characters are stored as binary data. When the value of a property is binary data (a Python byte-array) and you want to retrieve it as a string, then you must set `decode=True`.

```
data = esp.getProperty(name="中 employee_data 文",decode=True)
```

Clearing Properties

The `esp.clearProperty` removes the property from the Python context.

Table 9.10 *esp.clearProperty Function*

Syntax	Description
`esp.clearProperty(name=`*`name`*`[ ,field=`*`field`*`] [,context=`*`context`*`][,persist= true | false])`	Use `name` to specify the name of the property to clear. Use the `field` parameter to specify a field of the property to clear. Use `context` to specify a different Python context in which to clear the property. Use `persist` to specify whether to remove the property value from disk.

The `esp.clearProperty` function returns either **`true`** when the property is successfully cleared, or **`false`** when the ESP server encountered an error when attempting to clear the property.

Persistent Properties

Persistent properties are saved to the file system when they are set. The ESP server uses these saved values to initialize the property values upon start-up.

To enable persistent properties, set the `persist` property of the `esp.setProperty` or the `esp.clearProperty` function:

```
esp.setProperty(name="myproperty",value="foo",persist=True)

esp.clearProperty(name="myproperty",persist=True)
```

The properties are stored in a directory that is derived from the project's Lua root directory.

Manipulating Images

SAS Event Stream Processing provides functions to manipulate image data within Python code.

To use these functions, you must import the esp_utils package:

```
import esp_utils
```

Function	Description
`sas_wide_image_to_opencv_image`	Converts an image from SAS wide format to a format that you can use with OpenCV. By default, the `copy` parameter is set to **`True`**, which slows image processing and enables you to edit or annotate the image. Set `copy=False` to process the image more quickly.
`my_opencv_image = esp_utils.image_conversion.sas_wide_image_to_opencv_image(data['my_sas_wide_image'],`	
`opencv_image_to_sas_wide_image`	Converts an image that was created with OpenCV to SAS wide format. **Note:** It is often recommended to use SAS wide format to improve performance in certain scenarios because it avoids the overhead of image compression.

Function	Description
`my_sas_wide_image = esp_utils.image_conversion.opencv_image_to_sas_wide_image(data['my_opencv_image'])`	
`blob_image_to_opencv_image`	Converts a JPEG or PNG image to a format that you can use with OpenCV.
`my_opencv_image = esp_utils.image_conversion.blob_image_to_opencv_image(data['my_blob_image'])`	
`opencv_image_to_blob_image`	Converts an image that was created with OpenCV to a JPEG (default) or PNG image. Specify the output `type` as **`.jpeg`** (default) or **`.png`**. Specify an integer `quality` value between **`0`** and **`100`** to accommodate the number of frames you send per second. This gives you control over file size and quality. The default value is **`85`**.
`my_blob_image = esp_utils.image_conversion.opencv_image_to_blob_image(data['my_opencv_image'],type=".jp`	

Note: JPEG and PNG images are required for Grafana.

Working with Tensors

SAS Event Stream Processing provides functions to work with tensors within Python code. This topic is relevant to the esp_utils package. For more information about tensors, see "Representing Tensors in a Score Window" in *SAS Event Stream Processing: Using Streaming Analytics*

Use the `np_array_to_tensor` function to convert a NumPy array to an ESP tensor. If you do not intend to preprocess data for ONNX model scoring, you can use this function instead. You can prepare the data using NumPy, and then use this function to convert it into a tensor.

```
my_tensor = np_array_to_tensor(data['my_numpy_array'])
```

Use the `tensor_to_np_array` function to convert tensor output to a NumPy array. The resulting NumPy array can be useful to decode (postprocess) the output of an ONNX model.

```
my_numpy_array = tensor_to_np_array(data['my_tensor'])
```

Postprocessing

This topic is relevant to the esp_utils package.

Use the `bbox_letterbox_to_original` function to convert a `x,y,w,h` bounding box in a letterbox image to the coordinates in the original image. This function can be useful when you use a Computer Vision object detection model on a letterbox resized image, and you want to visualize the detected objects in the original (not resized) frame.

Use the `xy_letterbox_to_original` function to convert an `(x,y)` position in a letterbox image to the coordinates in the original image. This function can be useful when you use Computer Vision keypoint detection models, such that you can visualize the keypoints in the original frame.

Calling Functions from Python Snippets

This topic is relevant to the esp package. You can use the `esp.call` function to call functions that are defined in Python snippets. First, specify the parameters that you want to use as data within the called function. Next, the `_pyname_` parameter identifies the snippet to reference. The second `_pyfunc_`parameter identifies the function within that snippet to execute. The server removes these parameters and passes all others to the specified function.

```
response =
esp.call(parameter,parameter,_pyname_="snippet_name",_pyfunc_="func_nam
e")
```

The return value is a Python object that contains the following fields:

- `code` - a Boolean value that indicates whether the function was executed successfully.
- `results` - an array of the return values from the function if the call executed successfully.
- `error` - an error description if the call failed.

Consider the following snippet:

```
<python-snippet name="math">
  <code><![CDATA[

        import math

        def divide(dividend,divisor):
            value = dividend / divisor
            return value

        ]]></code>
</python-snippet>
```

You could call the `divide` function as follows:

```
import esp

def create(data,context):

    e = []
    dividend = data["dividend"]
    divisor = data["divisor"]

    response =
esp.call(dividend,divisor,_pyname_="snippet",_pyfunc_="divide")

    if (response["code"] == True):
        e["quotient"] = response["results"][0]
    else:
        e["error"] = response["error"]

    return e
```

If there is a division by 0, the code field of the response is **`false`** and the error field of the response indicates the error. Otherwise, the code field is **`true`** and the results field contains a single entry that provides the quotient.

For more information, see "Using Python Snippets".

Miscellaneous Functions

To use these functions, you must import the esp package:

```
import esp
```

Table 9.11 *Miscellaneous Functions*

Function	Description

`esp.getServerProperty(name)`	Retrieve the value of a server level property from the ESP server. The specified on the server command line, in esp-properties.yml, or in the example, if you wanted to send a REST request to get the ESP project the server in which you are running:
	`host = esp.getServerProperty("HOST")`
	`port = esp.getServerProperty("http")`
	`url = "http://" + host + ":" + port + "/eventStreamProcessing/v2/pro`
	
	`request = {}`
	`request["url"] = url`
	`request["method"] = "GET"`
	`request["headers"] = {"accept":"application/json"}`
	`request["topython"] = True`
	`response = esp.sendHttp(request)`
	`print(str(response))`

Function	Description

`esp.getProjectProperty(name, [project])`	Retrieve the value of a project level property. The project parameter i do not specify a project name, and there is a single project in the serv used. These values are defined in the project XML:
	`<project>`
	`  <properties>`
	`  <property name="property_1">value 1</property>`
	`  <property name="property_2">value 2</property>`
	`  </properties>`
	`  ...`
	`</project>`

Accessing Relational Databases from Python Components

Overview

You can access relational databases from Python components through ODBC drivers.

When using an ODBC driver, you establish connectivity to a database through a system data source name (DSN). A DSN and the associated database user credentials are required configuration parameters. The general format of a DSN is as follows:

```
DSN=data_source[;attribute=value[;attribute=value]...]
```

You must specify a *data_source*, which names the desired wire protocol to use for database connectivity. For the data source that you specify, you enumerate one or more relevant attributes. Most wire protocols require that you specify a *userid* and *password* in order to access the database. Many wire protocols also require that you specify a *hostname* that identifies the location of the database.

Each attribute that you specify for the *data_source* overrides the value set in the odbc.ini file that is supplied with SAS Event Stream Processing.

For more information, see "Using DataDirect ODBC Drivers" in *SAS Event Stream Processing: Connectors and Adapters*.

Database connections are stored in Python contexts. In most cases, you do not need to specify a context when you create a connection because a default context is created by the ESP server.

IMPORTANT Python component access to the following relational databases is supported.

- Db2
- Greenplum
- MySQL
- Oracle
- PostgreSQL

Use unsupported databases at your own discretion.

Connecting to a Database

Use the `esp.sqlConnect` function to connect to a database. For example:

```
code,err = esp.sqlConnect(options)
```

In the function, **`options`** is a Python object that contains the following fields:

- `name` is the name of the SQL connection.
- `connstr` is the DSN, which is described here.
- `context` is an optional value that specifies the Python context in which to create the connection.

The name of the connection is used in subsequent database calls.

The `esp_sqlConnect` function returns two values. The first value returned is a Boolean that indicates whether the connection was successful. If the connection failed, then the reason for failure is returned through the second value.

For example, the following code shows how to connect to a PostgreSQL database:

```
connstr = "DSN=PostgreSQL Wire
Protocol;UID=postgres;PWD=mypass123;servername=myserver;port=4444;datab
ase=espdemo"
code,err = esp.sqlConnect(name="db",connstr=connstr)
```

When you are finished with the connection, you can use the `esp.sqlDisconnect` function to remove it.

```
code,err = esp.sqlDisconnect(options)
```

In the function, **`options`** is a Python object that contains the following fields:

- `name` is the name of the SQL connection to disconnect.
- `context` is an optional value that specifies the Python context in which the connection exists.

The `esp.sqlDisconnect` function returns two values. The first value returned is a Boolean that indicates whether the disconnection was successful. If the

disconnection failed, then the reason for failure is returned through the second value.

The following code shows how to disconnect a Postgres connection:

```
code,err = esp.sqlDisconnect(name="db")
```

Retrieving Data

There are two ways to retrieve data from a database. The first way obtains all the results from an SQL query by sending an SQL select statement to the ESP server.

```
resp = esp.sqlSelect(options)
```

In the statement, **options** is a Python object that contains the following fields:

- `conn` is the name of the SQL connection to use.
- `query` is the SQL select statement.
- `context` is an optional value that specifies the Python context in which the connection exists.

The return value is a Python object that contains the following fields:

- `code` is a Boolean value that indicates whether the query was successful.
- `data` is the database data.
- `message` is an informative message sent when the query fails.

You can use `esp.sqlSelect` with moderately sized queries. For example, suppose millions of stock trades are stored in a trades table. The following code shows how to retrieve extremely large trades (of more than a million shares):

```
resp = esp.sqlSelect(conn="db",query="select * from trades where quant
> 1000000")

print(str(resp))
```

```
{'code': True,
    'data': (
        {'broker': 101667.0, 'buyer': 1012223.0, 'buysellflg': 0.0,
'currency': 87236.0, 'id': 2669875.0, 'msecs': 418.0, 'price': 4.15,
'quant': 2063400.0, 'seller': 8382739.0, 'symbol': 'C', 'udate':
1280928609000000.0, 'venue': 55555.0},
        {'broker': 101667.0, 'buyer': 0.0, 'buysellflg': 1.0,
'currency': 87236.0, 'id': 18564845.0, 'msecs': 724.0, 'price': 21.7,
'quant': 1286200.0, 'seller': 8123541.0, 'symbol': 'XEL', 'udate':
1280928728000000.0, 'venue': 55555.0}
)}
```

However, selecting all of the trades in one pass would consume too much memory. When you need to retrieve large amounts of data from the database, it is recommended to retrieve data in increments. Use the following functions to open a query and retrieve the data in increments:

- `esp_sqlOpen` - opens a query.

- `esp_sqlNext` - retrieves a specified number of rows from the result.
- `esp_sqlClose` - closes a query.

Use the `esp.sqlOpen` function as follows:

```
handle = esp.sqlOpen(options)
```

In the function, **`options`** include the following fields:

- `conn` is the name of the SQL connection to the ESP server.
- `query` is the SQL select statement.
- `context` is an optional value that specifies the Python context in which the connection exists.

The return value is a Python object that contains the following fields:

- `code` is a Boolean value that indicates whether the query was successful.
- `message` is an informative message that is sent when the query fails.

After you successfully open a query, you can retrieve the data with the `esp.sqlNext` function.

```
data,count = esp.sqlNext(handle,numrows)
```

- `handle` is the value that is returned from `esp.sqlOpen`.
- `numrows` is the number of rows to return.

If you do not specify a `numrows` parameter, then the remainder of the rows in the result are returned.

The function returns two values. The first value is the data from the query result, or **`nil`** when an error occurs. The second value is the number of records returned, or **`nil`** when an error occurs or when the data has been exhausted. The data is exhausted whenever the function returns a count of 0. When that happens, the server closes the query, so you do not need to call `esp.sqlClose`.

When you are finished with the query, use the `esp.sqlClose` function to remove the query from the server.

```
code,err = esp.sqlClose(handle)
```

The function returns two values. The first value is a Boolean that indicates whether the close was successful. The second value contains an error message that provides information about the cause of the failure.

As previously mentioned, when you have retrieved all the rows you do not need to close the query because the ESP server does it for you.

Here is an example that reads a large stock trade (100 trades at a time) from Postgres and prints the symbol and broker:

```
handle = esp.sqlOpen(conn="db",query="select * from trades")

if (handle["code"] == True):
    while True:
        trades,count = esp.sqlNext(handle,100)

        if (trades == None):
            break
```

```
        for trade in trades:
            print("symbol=" + trade["symbol"] + ",
broker="..trade["broker"])
```

Inserting Data

Use the `esp.sqlInsert` function to insert records into the database.

```
resp = esp.sqlInsert(options)
```

In the function, **`options`** include the following fields:

- `conn` is the name of the SQL connection to use.
- `table` is the database table in which to insert data.
- `data` contains one or more Python objects that contain the records to insert.
- `context` is an optional value that specifies the Python context in which the connection exists.

If you are processing stock trades, then you could use a Python subscriber connector on the Source window to grab incoming trades and write them to a database:

```
<window-source name="trades" insert-only="true">
    <schema>
        <fields>
            <field key="true" name="id" type="int64"/>
            <field name="symbol" type="string"/>
            <field name="currency" type="int32"/>
            <field name="time" type="int64"/>
            <field name="msecs" type="int32"/>
            <field name="price" type="double"/>
            <field name="quant" type="int32"/>
            <field name="venue" type="int32"/>
            <field name="broker" type="int32"/>
            <field name="buyer" type="int32"/>
            <field name="seller" type="int32"/>
            <field name="buysellflg" type="int32"/>
        </fields>
    </schema>
    <connectors>
        <connector class="python" name="sub">
            <properties>
                <property name="type">sub</property>
                <property name="code"><![CDATA[
                    def subscribe(data):
                        response =
esp.sqlInsert(conn="db",table="trades",data=data)
                        if (response.code == False):
                            print(str(response))
                  ]]></property>
            </properties>
        </connector>
```

```
    </connectors>
</window-source>
```

The return value from `esp.sqlInsert` is a Python object that contains the following fields:

- `code` is the status of the insert: **`true`** for success and **`false`** for failure.
- `message` contains information that is related to the failure when one occurs.
- `object` is the data for the failed record when a failure occurs.

Updating Data

Use the `esp.sqlUpdate` function to update the database.

```
resp = esp.sqlUpdate(options)
```

There are two ways that you can update the database through Python components. The first way is to send only the data that contains the fields to update. When you use this approach, the following fields are accepted in the `options` parameter:

- `conn` is the name of the SQL connection to use.
- `table` is the database table to update.
- `data` contains one or more Python objects that contain the update.
- `upsert` is an optional Boolean value that attempts an insert of the record when one does not exist. The default value is **`false`**.
- `context` is an optional value that specifies the Python context in which the connection exists.

The server derives the database key from the data, so the data must contain at least all key fields in each object. This update method can accept multiple Python objects in the function call. Each object generates an individual update request to the database.

For example, suppose you are processing stock trades and want to save the minimum and maximum prices for each symbol. Create the following table:

```
create table trades_data
(
    symbol text,
    min_price float,
    max_price float,
    range float,
    primary key (symbol)
);
```

The following code streams trades and keeps the current minimum and maximum price for each symbol:

```
resp = esp.sqlUpdate(conn="db",
                             table="trades_data",
                             upsert=True,
                             data=[
```

```
{"symbol":"ABC","min_price":100,"max_price":120,"range":20},

{"symbol":"XYZ","min_price":10,"max_price":20,"range":10}
                                ])
```

Note that the `upsert` value is **`true`**. As a result, the server adds the record when it does not exist.

The second way to perform an update is to add a single Python object with an SQL WHERE clause that identifies the database records to update. In the following code, the data from the request is applied to all records that match the WHERE clause:

```
resp = esp.sqlUpdate(conn="db",
                     table="trades_data",
                     data={"min_price":0,"max_price":0,"range":0},
                     where="1=1")
```

Any key fields in the data are ignored. The `trades_data` table from the previous example is initialized, zeroing previously existing values.

The return value from `esp.sqlUpdate` is a Python object that contains the following fields:

- `code` is the status of the update: **`true`** for success and **`false`** for failure.
- `message` is information that is related to the failure when one occurs.
- `object` is the data for the failed record when a failure occurs.

Deleting Data

Use the `esp.sqlDelete` function to delete records from a database.

```
resp = esp.sqlDelete(options)
```

In the function, **`options`** is a Python table that contains the delete properties.

There are two ways that you can delete records from the database through Python components. The first way is to send one or more Python objects that contain key field data of the records to delete. When you use this approach, the `options` parameter accepts the following fields:

- `conn` is the name of the SQL connection to use.
- `table` is the database table from which to delete records.
- `data` contains one or more Python objects that contain the key fields of the records to delete.
- `context` is an optional value that specifies the Python context in which the connection exists.

The following code deletes a few records from the `trades_data` table in the update section:

```
resp = esp.sqlDelete(conn="db",
```

```
                                    table="trades_data",
                                    data=[{"symbol":"ABC"},
{"symbol":"XYZ"}])
```

The second way to delete records is to add an SQL WHERE clause that identifies the database records to delete. The following code deletes all the records from `trades_data` that meet a certain condition (`max_price > 10`:

```
resp = esp.sqlDelete(conn="db",table="trades_data",where="max_price >
10")
```

The return value from `esp.sqlDelete` is a Python object that contains the following fields:

- `code` is the status of the delete: **`true`** for success and **`false`** for failure.
- `message` is information that is related to the failure when one occurs.
- `object` is the data for the failed record when one occurs.

Using Python Modules

Python modules are self-contained blocks of Python code that exist at the project level. You can import these modules into any Python entity in the project by using the name of the module.

Use the `python-module` element to define the snippet. For more information, see "python-module" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Use the `code` element to specify the code for the snippet.

Consider the following module:

```
<project ...>
  <python-modules>
    <python-module name="module">
        <code><![CDATA[
            import esp
            class Module
            # Python code
         ]]>
        </code>
    </python-module>
  </python-modules>
    ...
  <contqueries>
      <contquery>
          ...
          <window-python...>
           ...
             <code><![CDATA[
                     import esp
                     from module import Module
```

```
                    # Python code
                ]]>
               </code>
             </window-python>
           ...
         </contquery>
    </contqueries>
</project>
```

Suppose that you wanted a class to encapsulate the ESP server logging and make it available to any Python entity:

```
<python-module name="logger">
    <code><![CDATA[

import esp

class Logger:

    def __init__(self,context):
        self._context = context if context != None else "mylogger"

    def error(self,msg,line = None):

esp.logMessage(logcontext=self._context,message=msg,level="error",line=
line)

    def warn(self,msg,line = None):

esp.logMessage(logcontext=self._context,message=msg,level="warn",line=l
ine)

    def fatal(self,msg,line = None):

esp.logMessage(logcontext=self._context,message=msg,level="fatal",line=
line)

    def info(self,msg,line = None):

esp.logMessage(logcontext=self._context,message=msg,level="info",line=l
ine)

    def debug(self,msg,line = None):

esp.logMessage(logcontext=self._context,message=msg,level="debug",line=
line)

    def trace(self,msg,line = None):

esp.logMessage(logcontext=self._context,message=msg,level="trace",line=
line)

    ]]></code>
</python-module>
```

Suppose that you have a Python window that divides numbers coming into an event. If the divisor is 0, you want to log an ERROR message to the server with the Logger class from this module.

```
<window-python name="python" events="create">
  <schema>
    <fields>
      <field name="id" type="string" key="true"/>
      <field name="dividend" type="int32"/>
      <field name="divisor" type="int32"/>
      <field name="quotient" type="double"/>
    </fields>
  </schema>
  <copy>id,dividend,divisor</copy>
  <use>dividend,divisor</use>
  <code><![CDATA[
    from logger import logger

    def create(data):

        e = {}
        dividend = data["dividend"]
        divisor = data["divisor"]

        if (divisor == 0):
            logger = Logger("modules.example")
            logger.error("division by 0, " + str(dividend) + "/" +
str(divisor))
            return None

        e["quotient"] = dividend / divisor

        return(e)
  ]]></code>
</window-python>
```

The Python window imports the Logger class from the logger module. Errors appear in the server log whenever a division by 0 occurs.

```
{
    "level":"ERROR",
    "message":"{6c00732f-5d9d-4203-9c42-731d4e442b30}[XMLServer0001] division by
0, 31\/0",
    "properties":
    {
        "caller":"Python.cpp:1910",
        "logger":"modules.example",
        "thread":"00000033"
    },
    "timeStamp":"2023-09-13T16:19:04.891000-04:00"
}
```

Using Python Snippets

Python snippets are self-contained blocks of Python code that exist at the project level. Other Python instances can access properties that are specified within a Python snippet.

Use the `python-snippet` element to define the snippet. For more information, see "python-snippet" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Use the `code` element to specify the code for the snippet.

The code can be either inline or in an external file. If it is in an external file you can use the file attribute to point to the file.

```
<code file="mycode.py" />
```

Consider the following snippet:

```
<project ...>
  <python-snippets>
    <python-snippet name="access_token" init="initfunc"
destroy="cleanup">
        <code><![CDATA[

            import esp
            from pathlib import Path

            def initfunc():
                print("initializing snippet")

            def cleanup():
                print("cleaning up snippet")

            def load():
                token = Path("token.txt").read_text()
                esp.setProperty(name="token",value=token)

            load()

        ]]></code>
    </python-snippet>
  </python-snippets>
    ...
</project>
```

After the snippet runs initialization and cleanup functions, it opens a file named **`token.txt`** and reads a token from it. Then the snippet sets a property called `token` in the ESP server. This property can be accessed by other Python entities.

Note: Code that is not inside of a function is executed when the Python entity is created and loaded.

Consider the following Python connector. It uses the token set by the Python snippet to make a REST request:

```
<connector class="python" name="pub">
  <properties>
    <property name="type">pub</property>
    <property name="interval">5 seconds</property>
    <property name="code"><![CDATA[

        token = esp.getProperty(name="token")

        def publish():
            # use token to make REST request, parse response, and get
events
            return {"events":data, "done": False}

    ]]></property>
  </properties>
</connector>
```

Note: The `esp.getProperty` function takes the name of the Python property.

Python snippets also enable you to start a thread that calls a specified function at a specified interval. The following code revises the previous snippet so that when the token expires after a period, the snippet gets a new token every five minutes and sets the property.

```
<project ...>
  <python-snippets>
    <python-snippet name="largetrades">
        <python-thread func="load" interval="5 minutes" />
        <code><![CDATA[

            def load():
                from pathlib import Path
                contents = Path("token.txt").read_text()
                esp.setProperty(name="token",value=token)
                return True

            load()

        ]]></code>
    </python-snippet>
  </python-snippets>
  ...
</project>
```

The return code from the function is a Boolean variable that tells the ESP server whether to keep the thread running. When it returns **`True`**, the thread continues to run. When it returns **`False`**, the thread terminates.

Now, the Python connector can grab the token each time it publishes to ensure that it has a fresh token that works with its request:

```
<connector class="python" name="pub">
  <properties>
    <property name="type">pub</property>
    <property name="code"><![CDATA[

        def publish():
            local   token = esp.getProperty(name="token")
            # use token to make REST request, parse response, and get
events
            return {"events":data, "done": False}

    ]]></property>
  </properties>
</connector>
```

The code for Python snippets is loaded in the order in which the snippets are declared in the XML. This means a snippet that is declared before another snippet might not have the variables available when the code is loaded.

Consider the following code:

```
<python-snippets>
    <python-snippet name="thisSnippet">
        <code><![CDATA[

            esp.setProperty(name="theproperty",value="hello, world")

        ]]></code>
    </python-snippet>
    <python-snippet name="thatSnippet">
        <code><![CDATA[

            value = esp.getProperty(name="theproperty")
            print("VALUE IS: " + str(value))

        ]]></code>
    </python-snippet>
</python-snippets>
```

When these snippets are included as specified, the code returns the following output in the ESP log:

```
VALUE IS: hello, world
```

Suppose that the snippets are transposed:

```
<python-snippets>
    <python-snippet name="thatSnippet">
        <code><![CDATA[

            value = esp.getProperty(name="theproperty")
            print("VALUE IS: " + str(value))

        ]]></code>
    </python-snippet>
```

```
    <python-snippet name="thisSnippet">
        <code><![CDATA[

            esp.setProperty(name="theproperty",value="hello, world")

        ]]></code>
    </python-snippet>
</python-snippets>
```

The following output is returned to the ESP log because the code for `thatSnippet` is run before the code for `thisSnippet`:

```
VALUE IS: nil
```

Handling Exceptions in Python Code

Python code can throw exceptions. Exceptions from the `init` function cause the project to fail to load. Exceptions from the `create` function, the `heartbeat` function, and snippets (when threaded) are handled based on the `on-script-exception` attribute of the `project` element. For more information, see "project" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

10

Expressions

Overview to Expressions

You can use expressions in Expression Engine Language (EEL) to define the following:

- filter conditions in Filter windows
- non-key field calculations in Compute, Aggregate, and Join windows
- matches to pattern logic in events of interest (EOIs) in Pattern windows
- window-output splitter-slot calculations (for example, you can use an expression to evaluate where to send a generated event)

Note: In EEL code, you must escape any special character within regular expressions using the \\ characters.

You can use functions written in Lua or Python instead of expressions in all of these cases except non-key field calculations in Compute, Aggregate, and Join windows.

You can use user-defined functions written in C instead of expressions in all of these cases except for pattern matching.

For information about how to specify expressions, refer to the *Expression Language: Reference Guide*.

Writing and registering expressions with their respective windows can be easier than writing the equivalent user-defined functions in C. However, expressions run

more slowly than functions. For very low-latency applications, user-defined functions minimize the overhead of expression parsing and processing.

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

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

To initialize expression engines for expression windows and window splitters, use the `<expr-initialize>` element in your XML code. For example, the following XML code initializes a user-defined function that is written in EEL for a splitter:

```
<expr-initialize>
        <udfs>
         <udf name='udf1' type='int32'>
          <![CDATA[private integer p
                   p = parameter(1);
                   return p%2]]>
         </udf>
        </udfs>
       </expr-initialize>
```

Using Expression Engine Language (EEL)

Understanding Data Type Mappings

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

The following table shows the supported data type mappings.

Table 10.1 *Expression Data Type Mappings Table*

Event Stream Processing Expressions	Expressions	Notes and Restrictions
String (utf8)	String (utf8)	Strings passed to expressions might be truncated to 1024

Event Stream Processing Expressions	Expressions	Notes and Restrictions
		bytes. Use the window-specific attribute `exp-max-string= 'N'` or the C++ window method `dfESPwindow::set_EXP_strMax(int N)` to set the maximum size of expression strings for a window.
date (second granularity)	date (second granularity)	Seconds granularity
timestamp (microsecond granularity)	date (microsecond granularity)	Constant milliseconds in dfExpressions not supported
Int32 (32 bit)	Integer (64 bit)	64-bit conversion for dfExpressions
Int64 (64 bit)	Integer (64 bit)	64-bit, no conversion
double (64 bit IEEE)	real (192 bit fixed decimal)	real 192-bit fixed point, double 64-bit float
money (192 bit fixed decimal)	real (192 bit fixed decimal)	192-bit fixed point, no conversion

Using Event Metadata in Expressions

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

Table 10.2 *Reserved Words to Access an Event's Metadata*

Reserved Word	Opcode
ESP_OPCODE Use this reserved word to obtain the opcode of an event in a given window.	I — Insert U — Update P — Upsert

Reserved Word	Opcode
	D— Delete SD — Safe Delete A safe delete does not generate a “key not found” error. **Note:** Values are case-sensitive.
ESP_FLAGS Use this reserved word in expressions to get the flags of an event in a given window.	N — Normal P — Partial R — Retention Delete

Using Expression Engine Language Global Functions

The Expression Engine Language supports global functions, also called user-defined functions (UDFs). You can register them as global functions and reference them from any expression window or window splitter expression.

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

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

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

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

Using SAS Data Quality Functions

Event stream processing expressions support the use of SAS Data Quality functions. The following functions are fully documented later in this document:

- dq.case
- dq.extract

- dq.gender
- dq.getlasterror
- dq.identify
- dq.initialize
- dq.loadqkb
- dq.matchcode
- dq.matchscore
- dq.parse
- dq.pattern
- dq.standardize
- dq.token
- dq.tokenvalue
- dq.value

To use these functions with SAS Event Stream Processing, you must set two environment variables as follows:

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

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

IMPORTANT To use SAS Data Quality software with SAS Event Stream Processing in a Kubernetes environment, the software must be installed on a persistent volume available to SAS Event Stream Processing.

11

Dates and Times

How Time Is Saved and Processed

SAS Event Stream Processing saves all time values relative to the UNIX epoch time (00:00 UTC on January 1, 1970). Stamp data is processed at a granularity of 1 microsecond. Date data is processed at a granularity of 1 second. The time is saved as an int64 value.

How Dates and Times Are Handled

Connectors and Adapters

Connectors and adapters can convert between a human readable timestamp, such as 01/01/2019 09:30:30 AM and the UNIX epoch time. For connectors and adapters that can convert time, a `dateformat` key exists that describes the format of the date or time value. For connectors and adapters that are written in C++, this format is in `strftime`. For adapters written in Java, this format is in Java 8 SimpleDateFormat.

For fields that are defined as a `timestamp`, microsecond precision is supported. Neither `strftime` nor SimpleDateFormat support microseconds with the format string. For times with a seconds value that includes a period, the value after the period is converted to microseconds.

For example, consider the following timestamp:

```
10/16/2019 16:17:30.123456
```

For a connector or an adapter written in C++, the `dateformat` string is as follows:

```
"%m/%d/%Y %H:%M:%S"
```

For an adapter written in Java, the `dateformat` string is as follows:

```
"mm/DD/yyyy HH:mm:ss"
```

The timestamp `10/16/2019 16:17:30.123456` is converted into the microsecond value 1571242650123456.

All date and time data is converted to Coordinated Universal Time (UTC), no matter how time-stamped by its source. UTC is a successor to Greenwich Meridian Time (GMT).

Windows

Whenever a window processes an event that contains the time data, the raw data is in UNIX epoch UTC format. Any window can format the raw data as you choose.

Expression Engine Language

The Expression Engine Language (EEL) uses a different time format, saving the value as a number. The number represents days since 12/30/1899. Hours, minutes, seconds, and milliseconds are converted to a fraction, where 1 hour = 1/24 units, 1 minute = 1/(24*60) units, and so on.

SAS Event Stream Processing converts between the UNIX epoch format and the expression engine language format when date or time values are used. To modify the value in the expression engine language, use the expression engine language format.

For example, the following expression subtracts an hour from the input date:

```
return (input_date - (1.0/24.0))
```

To remain consistent with SAS Event Stream Processing times, use the `todayGMT()` function instead of the `today()` function when you create a date field in the expression engine language.

SAS Micro Analytic Service Modules

When you use SAS Micro Analytic Service modules with DS2 or Python, the date or time value is not converted into native formats. Instead, the value is converted into SAS epoch format, which starts at January 1, 1960. The value is passed as a BIGINT for DS2 and a long for Python. For more information, see "Understanding Data Type Mappings".

The difference between the UNIX epoch and SAS epoch times is as follows:

- EPOCH_DIFF_SECONDS: 315619200
- EPOCH_DIFF_MICROSECONDS: 315619200000000

Thus, in a Python program that obtains the UNIX epoch time from an input date value, subtract 315619200 from that value.

You can use the following general functions to access or manipulate data and time data in Functional and Notification windows.

Table 11.1 *Time Functions*

Function	Description
EVENTTIMESTAMP	Returns the timestamp of the current event
TIMECURRENT	Returns the current time.
TIMEDAYOFMONTH	Returns the day of the month of the current or specified time.
TIMEDAYOFWEEK	Returns the day of the week of the current or specified time.
TIMEDAYOFYEAR	Returns the day of the year of the current or specified time.
TIMEGMTTOLOCAL	Converts the GMT that is specified in the argument to local time.
TIMEGMTSTRING	Writes the GMT time represented by the first argument.
TIMEHOUR	Returns the hour of the day of the current or specified time.
TIMEMICRO	Returns the number of microseconds of the supplied argument, or the number of microseconds since Jan 1, 1970 when an argument is not specified.

Function	Description
TIMEMILLI	Returns the number of milliseconds of the supplied argument, or the number of microseconds since Jan 1, 1970 when an argument is not specified.
TIMEMINUTE	Returns the minute of the hour of the current or specified time.
TIMEMINUTEOFDAY	Returns the minute of the day of the current or specified time.
TIMEPARSE	Returns a string that represents the time specified in the first argument.
TIMESECOND	Returns the second of the minute of the current or specified time.
TIMESTAMP	Returns the current time as a string.
TIMESTRING	Returns the time represented by the first argument.
TIMESECONDOFDAY	Returns the second of the day of the current or specified time.
TIMETODAY	Returns a value that represents the first second of the current day relative to local time.
TIMEYEAR	Returns the number of years since 1900 of the current or specified time.

12

Array Functions

Array Functions

In the Expression Engine Language (EEL), it is possible to create arrays of simple types such as string, integer, date, Boolean, and real. Currently there are three functions that apply to array types: DIM, GET, and SET.

Dictionary

DIM Function

Creates, resizes, or determines the size of an array. If a parameter is specified, the array is resized or created. The new size is returned.

Category: Array

Returned data type: Integer

Syntax

arrayName.**DIM**<(*newsize*)>

Required Argument

arrayName
is the name of the array that you declared earlier in the process.

Optional Argument

newsize
is the optional numeric size (dimension) of the array. This can be specified as a numeric constant, field name, or expression.

Details

The DIM function is used to size and resize the array. It creates, resizes, or determines the size of the array. If a parameter is specified, the array is created or resized. The supported array types include:

- String
- Integer
- Date
- Boolean
- Real

Example

```
// declare the string array
string array string_list
// Set the dimension of the String_List array to a size of 5
rc = string_list.dim(5) // outputs 5
// <omitted code to perform some actions on the array>
// Re-size the array size to 10
rc = string_list.dim(10) // outputs 10
// Query the current size
re = string_list.dim() // outputs 10
```

GET Function

Retrieves the value of the specified item within an array. The returned value is the value of the array.

Category: Array

Returned data type: Integer

Syntax

array name.**GET**(<*n*>)

Required Argument

array name
is the name of the array that you declared earlier in the process.

Optional Argument

n
is the index of the array element for which the content is retrieved. This can be specified as a numeric constant, field name, or expression.

Details

The GET function returns the value of a particular element in the array.

Examples

Example 1

```
// Declare the string array "string_list" and Integer "i"
string array string_list
integer i

// Set the dimension of string_list array to 5 and initialize the
counter (i) to 1
string_list.dim(5)
i=1

// Set and print each entry in the array, incrementing the counter by 1
while(i<=5)
begin
   string_list.set(i,"Hello")
   print(string_list.get(i))
   i=i+1
end
```

Example 2

```
string array string_list
integer i

// set the dimension
string_list.dim(5)
i=1

// set and print each entry in the array
while(i<=5)
begin
     string_list.set(i,"Hello")
     print(string_list.get(i))
     i=i+1
end

// resize the array to 10
string_list.dim(10)
while(i<=10)
begin
     string_list.set(i,"Goodbye")
     print(string_list.get(i))
     i=i+1
end
```

SET Function

Sets values for items within an array. The returned value is the old value of the specified element in the array.

Category: Array

Returned data type: Integer

Syntax

array name.**SET**(<*n*,*"string"*>)

Required Argument

array name
is the name of the array that you declared earlier in the process.

Optional Arguments

n
is the number of the dimension that you are setting the value for; this can be specified as a numeric constant, field name, or expression.

"string"
is the value that you want to place into the array element; this can be specified as a string constant, field name, or expression.

Details

The SET function sets the value of an entry in the array.

Examples

Example 1

```
// Declare the string array "string_list"
// Set the dimension of string_list array to 5
string array string_list
string_list.dim(5)

// Set the first string element in the array to "Hello"
string_list.set(1,"Hello")
```

Example 2

```
string array string_list
string_list.dim(5)
// sets the first string element in the array to hello
string_list.set(1,"hello")
```

13

Data Quality Functions

Data Quality Functions

Expression Engine Language (EEL) supports the data quality object. You can use data quality to perform the listed functions (object methods) from within the EEL node. Some of the advantages of using data quality functions within the EEL include dynamically changing match definitions, reading match definitions from another column, or setting different definitions.

Dictionary

DQ.CASE Function

Applies casing rules (upper, lower, or proper) to a string. The function also applies context-specific casing logic using a case definition in the SAS Quality Knowledge Base (QKB).

Category: Data Quality

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.CASE(case_def, casing_type, input, result)

Required Arguments

casing_type
integer numeric constant that specifies the type of casing that is applied, [1 = uppercase, 2 = lowercase, 3 = proper case].

input
a string representing the input value or input field name.

result
a string representing the output field name.

Optional Argument

case_def
a string representing the name of a case definition in the QKB. Pass an empty string to use the default casing algorithm.

Details

The DQ.CASE function applies casing rules to an input string and outputs the result to a field.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized using a call to the function DQ_INITIALIZE.

You can specify one of three casing types: uppercase, lowercase, or propercase. When uppercase or lowercase is specified, the function applies Unicode uppercase or lowercase mappings to the characters in the input string. When propercasing is specified, the function applies uppercase mappings to the first letter in each word and lowercase mappings to the remaining letters.

The caller can invoke the use of a case definition. A case definition is an object in the QKB that contains context-specific casing logic. For example, a case definition implemented for the purpose of propercasing name data can be used to convert the string "Mcdonald" to "McDonald". Refer to the QKB documentation for information about what case definitions are available in your QKB. If you do not want to use a case definition, you can omit the case definition name by entering a blank string for the case definition parameter. In this case, generic Unicode case mappings are applied to the input string as described earlier.

Note: If you want to use a case definition, you must call DQ.LOADQKB before calling DQ.CASE. The function DQ.LOADQKB loads the contents of a QKB into memory and links that QKB with the data quality object. This enables DQ.CASE to access the case definition that you specify.

Example

```
dq dataq
 string output
 dataq = dq_initialize()
 dataq.case("", 1, "ronald mcdonald", output)
 // outputs "RONALD MCDONALD"

dataq.case("", 3, "ronald mcdonald", output)
 // outputs "Ronald Mcdonald"

dataq.loadqkb("ENUSA")
 dataq.case("Proper (Name)", 3, "ronald mcdonald", output)
 // outputs "Ronald McDonald"
```

DQ.EXTRACT Function

Extracts attributes from a string.

Category:	Data Quality
Returned data type:	Character
Note:	The returned value is a value, token, or token value from the extract function.

Syntax

DQ.EXTRACT*definition, string*

Required Arguments

definition
a string representing the name of the extraction definition in the QKB.

string
a string that represents the attribute that needs to be extracted.

Details

The DQ.EXTRACT function extracts attributes from a string into tokens. The first parameter is the name of the QKB extraction definition. The second is the string where the attributes are extracted. This function returns a number of tokens that were created. It returns 0 if it fails.

Example

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using
QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

DQ.GENDER Function

Determines the gender of an individual's name using a gender analysis definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.GENDER(*gender_def, input, result*)

Required Arguments

gender_def
a string representing the name of a gender analysis definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.GENDER function analyzes a string representing an individual's name and determines the gender of the name.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized using a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.GENDER, you must specify the name of a gender analysis definition. A gender analysis definition is an object in the QKB that contains reference data and logic used to determine the gender of the input name string. See your QKB documentation for information about which gender analysis definitions are available in your QKB.

Example

```
dq dataq
 string output
dataq = dq_initialize()
dataq.loadqkb("ENUSA")
dataq.gender("Name", "John Smith", output)
 // outputs "M"

 dataq.gender("Name", "Jane Smith", output)
 // outputs "F"

 dataq.gender("Name", "J. Smith", output)
 // outputs "U" (unknown)
```

DQ.GETLASTERROR Function

Returns a string describing the most recent error encountered by a data quality object.

Category:	Data Quality
Returned data type:	Character
Note:	The returned value is a string containing an error message.

Syntax

DQ.GETLASTERROR(<>)

Details

The DQ.GETLASTERROR function is a member of the data quality class. It returns an error message describing the most recent error encountered by a data quality object. The error might have occurred during invocation of any other data quality member function.

A best practice for programmers is to check the result code for each data quality call. If a result code indicates failure, use DQ.GETLASTERROR to retrieve the associated error message.

Example

```
dq dataq
 integer rc
```

```
 string errmsg
dataq = dq_initialize()
 rc = dataq.loadqkb("XXXXX")
 // an invalid locale name -- this will cause an error

 if (rc == 0) then
      errmsg = dq.getlasterror()
 // returns an error message
```

DQ.IDENTIFY Function

Identifies the context of a string using an identification analysis definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.IDENTIFY(*ident_def, input, result*)

Required Arguments

ident_def
a string representing the name of an identification analysis definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.IDENTIFY function analyzes a string and determines the context of the string. The context refers to a logical type of data, such as name, address, or phone.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized using a call to the function DQ.INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.IDENTIFY, you must specify the name of an identification analysis definition. An identification analysis definition is an object in the QKB that contains reference data and logic used to identify the context of the input string. Refer to your QKB documentation for information about which identification analysis definitions are available in your QKB.

Note: For each identification analysis definition, there is a small set of possible contexts that might be output. Refer to the description of an identification analysis definition in the QKB documentation to see which contexts that definition is able to identify.

Example

```
dq dataq
 string output
 dataq = dq_initialize()
dataq.loadqkb("ENUSA")
dataq.identify("Individual/Organization", "John Smith", output)
 // outputs "INDIVIDUAL"

dataq.identify("Individual/Organization", "DataFlux Corp", output)
 // outputs "ORGANIZATION"
```

DQ_INITIALIZE Function

Instantiates and initializes a data quality object.

Category:	Data Quality
Returned data type:	Character
Note:	The returned value is an initialized instance of a data quality object.

Syntax

DQ_INITIALIZE(<>)

Details

The DQ_INITIALIZE instantiates and initializes a data quality object. The object can then be used to invoke data quality class functions.

Example

```
dq dataq
dataq = dq_initialize()
```

DQ.LOADQKB Function

Loads definitions from a QKB into memory and links those definitions with the data quality object.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.LOADQKB(*locale*)

Required Argument

locale
a five-character locale code name representing a locale supported by the QKB.

Details

The function DQ.LOADQKB is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The function DQ.LOADQKB can be called after the initialization.

The DQ.LOADQKB function loads definitions from a QKB into memory and links those definitions with the data quality object. A definition is a callable object that uses context-sensitive logic and reference data to perform analysis and transformation of strings. Definitions are used as parameters in other dq functions.

When calling DQ.LOADQKB, you must specify a locale code. This locale code is a five-character string representing the ISO codes for the locale's language and country. Refer to your QKB documentation for a list of codes for locales that are supported in your QKB.

Note: Only one locale code can be specified in each call to DQ.LOADQKB. Only definitions associated with that locale are loaded into memory. This means that support for only one locale at a time can be loaded for use by a data quality object. In order to use QKB definitions for more than one locale, you must either use

multiple instances of the data quality class or call DQ.LOADQKB multiple times for the same instance, specifying a different locale with each call.

Example

```
dq dataq_en
 // we instantiate two dq objects

dq dataq_fr
 string output_en
 string output_fr
dataq_en = dq_initialize()
dataq_fr = dq_initialize()

 dataq_en.loadqkb("ENUSA"
 // loads QKB support for locale English, US

 dataq_fr.loadqkb("FRFRA")
 // loads QKB support for locale French, France

 dataq_en.gender("Name", "Jean LaFleur", output_en)
 // output is 'U'

dataq_fr.gender("Name", "Jean LaFleur", output_fr)
 // output is 'M'
```

DQ.MATCHCODE Function

Generates a match code for a string using a match definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.MATCHCODE(*match_def, sensitivity, input, result*)

Required Arguments

match_def
a string representing the name of a match definition in the QKB.

sensitivity
integer numeric constant that specifies the sensitivity level to be used when generating the match code [possible values are 50–95].

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.MATCHCODE function generates a match code for an input string and outputs the match code to a field. The match code is a fuzzy representation of the input string. It can be used to do a fuzzy comparison of the input string to another string.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.MATCHCODE, you must specify the name of a match definition. A match definition is an object in the QKB that contains context-specific reference data and logic used to generate a match code for the input string. Refer to your QKB documentation for information about which match definitions are available in your QKB.

You must also specify a level of sensitivity. The sensitivity indicates the level of fuzziness that is used when generating the match code. A higher sensitivity means that the match code is less fuzzy (yielding fewer false positives and more false negatives in comparisons). A lower sensitivity means that the match code is more fuzzy (yielding fewer false negatives and more false positives in comparisons). The valid range for the sensitivity parameter is 50–95.

Example

```
dq dataq
 string output
dataq = dq_initialize()
 dataq.loadqkb("ENUSA")
 dataq.matchcode("Name", 85, "John Smith", output)
 // Outputs match code "4B~2$$$$$$$C@P$$$$$$"

dataq.matchcode("Name", 85, "Johnny Smith", output)
 // Outputs match code "4B~2$$$$$$$C@P$$$$$$"
```

DQ.MATCHSCORE Function

Processes two input strings along with the name of the match definition and outputs the sensitivity for the match strings.

Category: Data Quality

Syntax

DQ.MATCHSCORE(*definition_name, input1, input2, use_wildcards*)

Required Arguments

definition_name
the name of the match definition to use.

input1
the first input string to check.

input2
the second input string to check.

returns
the sensitivity value.

use_wildcards
true if wildcards in generated match codes should be considered for the purpose of scoring.

Details

The DQ.MATCHSCORE function determines the highest sensitivity value where two input strings generate the same match code.

Example

```
dq dataq;
dataq = dq_initialize();
dataq.loadqkb("ENUSA");

integer x;
x = dataq.matchscore("Name", "John Jones", "John J. Jones", false);
```

DQ.PARSE Function

Parses a string.

Category: Data Quality

Syntax

DQ.PARSE(*definition, string*)

Required Arguments

definition
a string representing the parsed data.

returns
the number of tokens.

string
the input string to be parsed into tokens.

Details

The DQ.PARSE function parses the input string into tokens. The first parameter is the name of the QKB parse definition. The second parameter is the string from which the tokens are parsed. This returns the number of tokens created. It returns 0 if it fails.

Example

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
```

```
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.PATTERN Function

Generates a pattern for a string using a pattern analysis definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.PATTERN(*pattern_def, input, result*)

Required Arguments

pattern_def
a string representing the name of a match definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.PATTERN function generates a pattern for the input string and outputs the pattern to a field. The pattern is a simple representation of the characters in the input string. Such patterns can be used to perform pattern frequency analysis for a set of text strings.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.PATTERN, you must specify the name of a pattern analysis definition. A pattern analysis definition is an object in the QKB that contains logic used to generate a pattern for the input string. Refer to your QKB documentation for information about which pattern analysis definitions are available in your QKB.

Example

```
dq dataq
 string output
dataq = dq_initialize()
 dataq.loadqkb("ENUSA")
 dataq.pattern("Character", "abc123", output)
 // Outputs "aaa999"
```

DQ.STANDARDIZE Function

Generates a standard for a string using a standardization definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.STANDARDIZE(*stand_def, input, result*)

Required Arguments

stand_def
a string representing the name of a standardization definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.STANDARDIZE function generates a normalized standard for an input string and outputs the standard to a field.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.STANDARDIZE, you must specify the name of a standardization definition. A standardization definition is an object in the QKB that contains context-specific reference data and logic used to generate a standard for the input string. Refer to your QKB documentation for information about which standardization definitions are available in your QKB.

Example

```
dq dataq
 string output
 dataq = dq_initialize()
dataq.loadqkb("ENUSA")
dataq.standardize("Name", "mcdonald, mister ronald", output)
 // Outputs "Mr Ronald McDonald"
```

DQ.TOKEN Function

Obtains a token name for the index from a parse or extract function.

Category: Data Quality

Returned data type: Character

Note: This function returns true on success and false if the index is out of range. The token name is returned in the second parameter from the parse or extract function.

Syntax

DQ.TOKEN(integer, string)

Required Arguments

integer
the index of the token for which the name is desired.

string
the output string that receives the token name.

Details

The DQ.TOKEN function is used to retrieve an extraction or parse token name for the index. This function follows a parse or extract function.

Examples

Example 1

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using
QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

Example 2

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.TOKENVALUE Function

Obtains an attribute from the last parse or extract function.

Category: Data Quality

Returned data type: Character

Note: The returned value is a token value from the parse or extract function. Returns true if the token value is found or false if not.

Syntax

DQ.TOKENVALUE(<*token string, output string*>)

Required Arguments

token string
returns true if the token is found and false if the token is not found.

output string
a string that represents the value for that token.

Details

The DQ.TOKENVALUE function is used to retrieve an extraction or parse result value. The first parameter is the token name. It returns the attribute stored in that token in the second parameter. It returns true on success and false on failure (for example, if the token was not found). This function follows a parse or extract function.

Examples

Example 1

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
```

```
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using
QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

Example 2

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.VALUE Function

Retrieves the value from the last parse or extract function.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is true if the index is valid.

Syntax

DQ.VALUE(*integer, string*)

Required Arguments

integer
represents an index of the token.

string
a string that represents the output value.

Details

The DQ.VALUE function is used to retrieve an extraction or parse result. The first parameter is the index of a token. The second parameter receives the attribute that is stored in the token. The function returns true if it is able to get the token value. It returns false if it fails. This function follows a parse or extract function.

Examples

Example 1

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using
QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
```

```
print ("Colors = " & output)
```

Example 2

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dataq.token(i, output)
     print ("token #" & i & " = " & output)
     dataq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

14

Date and Time Functions

Overview

Dates, along with integers, reals, Booleans, and strings, are considered basic data types in the EEL. Similar to other basic data types, EEL provides functions to perform operations on dates.

Dictionary

FORMATDATE Function

Returns a date formatted as a string.

Category:	Date and Time
Returned data type:	String
Note:	The returned value is a string with the date formatted as a string.

Syntax

FORMATDATE(<datevar>,<format>)

Required Arguments

<datevar>
a date that needs to be formatted; this can be specified as field name.

<format>
a string that represents the format that needs to be applied; this can be specified as fixed string, field name, or expression.

Details

The format parameter can include any string, but the following strings are replaced with the specified values:

- YYYY: four-digit year
- YY: two-digit year
- MMMM: full month in proper case
- MMM: abbreviated three-letter month
- MM: two-digit month
- DD: two-digit day
- hh: hour
- mm: minute
- ss: second
- ms: millisecond

Note: The format parameters are case sensitive.

The FORMATDATE function dates should be in the format specified by ISO 8601 (YYYY-MM-DD hh:mm:ss:ms) to avoid ambiguity. Remember that date constants must start with and end with the # sign (for example, #12-February-2010#).

Examples

Example 1

```
// Declare a date variable and initialize
// it to a value
```

```
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dteval,"MM/DD/YY")
Results: 02/12/10
```

Example 2

```
// Declare a date variable and initialize
// it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval,"DD MMM YYYY")
Results: 12 Feb 2010
```

Example 3

```
// Declare a date variable and initialize
// it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval,"MMMM DD,YYYY")
Results: February 12, 2010
```

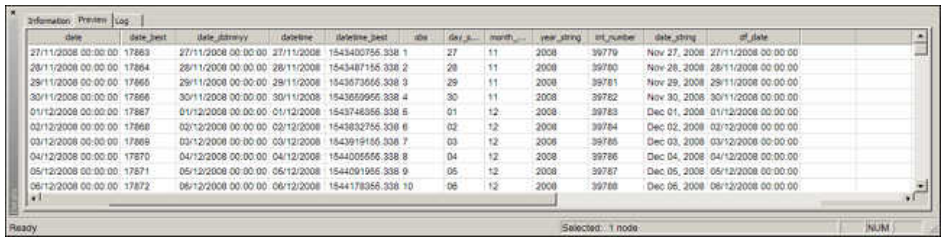
Example 4

```
day_string=formatdate('date',"DD");
month_string=formatdate('date',"MM");
year_string=formatdate('date',"YYYY");

int_number='date';
date_string=formatdate(int_number,"MMM DD,YYYY");

df_date='date';
```

Output 14.1 *Results*



TODAY Function

Returns the current data and time. This function is based on the local time zone.

Category:	Date and Time
Returned data type:	Character
Note:	The returned value is a date that represents the current date and time value.

Syntax

TODAY(< >)

Details

The TODAY function returns the current date and time value. For example, at 4:00 p.m. on February 12, 2010, the function would return the value "02/12/10 4:00:00 PM". Although it is represented as a character string, the actual value is a date value. For more information, see Date Expressions.

Before using this function, you must first declare a date variable to contain the date/time value.

Example

```
// declare the date variable to contain the date and time
date currentdate
// Use the TODAY function to populate the date variable with the
current date and time
currentdate = TODAY()
```

TODAYGMT Function

Returns the current date and time. This function is based on Greenwich Mean Time (GMT).

Category:	Date and Time
Returned data type:	Character
Note:	The returned value is a date and time combination that represents the current GMT date and time value.

Syntax

TODAYGMT(< >)

Details

The TODAYGMT function returns the current GMT date and time value. For example, at 4:00 p.m. on February 12, 2010, the function would return the value "02/12/10 4:00:00 PM". Although it is represented as a character string, the actual value is a date value.

Before using this function, you must first declare a date variable to contain the date/time value.

Example

```
// declare the date variable to contain the date and time
date currentdate
// Use the today function to populate the date variable with the
current date and time
currentdate = todaygmt()
```

15

File Functions

Overview

Use the external file object to work with files in the Expression Engine Language (EEL). Read and Write operations are supported in the file object and there are additional functions for manipulating and working with files.

Overview of the File Object

The file object can be used to open, read, and write files. A file is opened using the file object. For example:

```
File f
f.open("c:\filename.txt","r")
```

In this example, the OPEN() function opens filename.txt. The mode for opening the file is read. Other modes are "a" (append to end of file) and "w" (write). A combination of these switches can be used.

Executing Programs and File Commands

To execute programs, use the EXECUTE() function:

```
execute(string)
```

For example, the following code changes the default permissions of a text file created by the Data Job Editor.

To execute the command in Microsoft Windows, enter:

```
execute("/bin/chmod", "777", "file.txt")
```

Or, to execute from the UNIX shell, enter:

```
execute("/bin/sh", "-c", "chmod 777 file.txt")
```

Running a Batch File By Using Execute Function

To invoke the MS-DOS command prompt, call the cmd.exe file. For example:

```
//Expression
execute("cmd.exe", "/q" ,"/c", C:\BatchJobs.bat");
```

The following parameters can be declared:

- /q — turns echo off
- /c — Executes the specified command for the MS-DOS prompt and then closes the prompt

Note: The expression engine handles the backslash character differently; it does not need to be escaped.

For example: "C:\\Program Files\DataFlux" should now be entered as "C:\Program Files\DataFlux"

Dictionary

CLOSE Function

Closes an open file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**CLOSE**

Details

The CLOSE method closes the file that is currently open file (which was opened by using a fileobject.OPEN call) is closed.

Example

```
file myfile
if ( myfile.open("data.txt") ) then ...
rc = myfile.close()
```

COPYFILE Function

Copies a file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

COPYFILE(<source_file, target_file>)

Required Arguments

source_file
a string representing the name of the file to be copied; this can be specified as a fixed string, field name, or expression.

target_file
a string representing the name of the file to be written; this can be specified as a fixed string, field name, or expression.

Details

The COPYFILE function copies a file. If the target file exists, the COPYFILE function overwrites the target file.

Example

```
string source_file
string target_file
boolean rc_ok

source_file="C:\mydata.txt"
target_file="C:\mydata_copy.txt"

rc_ok = copyfile(source_file, target_file)
```

DELETEFILE Function

Deletes the specified file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DELETEFILE(<filename>)

Required Argument

filename
a string representing the name of the file to be deleted; this can be specified as a fixed string, field name, or expression.

Details

The DELETEFILE function deletes a file from disk. If the file did not exist, then the return code will be set to false.

Example

```
string filename
boolean rc_ok

filename="C:\mydata_copy.txt"

rc_ok = deletefile(filename)
```

EXECUTE Function

Runs the specified program.

Category:	External File
Returned data type:	Integer
Note:	The returned value represents the existing status of the program. If an error occurs, such as the program not found, then -1 is returned.

Syntax

EXECUTE(<filename<option1, option2,..., <option N>>

Required Argument

filename
a string representing the file (or command) to be executed; this can be specified as a fixed string, field name, or expression.

Optional Argument

option1...N
[optional] a string representing options that are passed to the file (command) that is going to be executed; this can be specified as a fixed string, field name, or expression.

Details

The EXECUTE function invokes a file (or operating system command).

Note: Use single quotation marks for any parameter with embedded spaces, as shown in the following example:

```
execute('cmd.exe','/C',
     '"C:\Progra~1\SASHome\DataFluxDataManagementStudio\2.9\bin\dmpexec
-j C:\ProgramData\DataFlux\DMStudio\studio1\MyRepo\FILEST~1\batch_jobs
\sample.djf
-l c:\temp\JobLogs\sample.log.txt"')
```

Note: The first parameter of the EXECUTE function should be the full path to the executable, script, or batch file. Do not quote or include any of the arguments. Pass the arguments in subsequent parameters to the EXECUTE function.

```
execute('path\to\executable.exe')
execute('executable.exe','argument')
```

Example

```
integer rc

// Windows example
rc = execute("cmd.exe", "/Q", "/C", "C:\mybatchjob.bat")
// /Q turns echo off
// /C executes the command specified by filename and then closes the
prompt

// Unix example
rc = execute("/bin/sh", "-c", "chmod 777 file.txt")
```

FILEEXISTS Function

Checks whether a specified file exists.

Category: External File

Returned data type: Character

Note: The returned value is true if the file exists.

Syntax

FILEEXISTS(<filename>)

Required Argument

filename
the name of the file that you are checking to find out if it exists.

Example

```
string filename
boolean rc_ok

filename="C:\doesexist.txt"

rc_ok = fileexists(filename) // outputs "true" if file exists
```

MKDIR Function

Creates a directory.

Category: String

Returned data type: Boolean

Syntax

MKDIR(*string*<, *create-intermediary-directories*>)

Required Argument

string
specifies the text string that contains the directory to create.

Optional Argument

create-intermediary-directories
specifies whether to create intermediary directories if they do not exist, using these values:

TRUE	specifies to create the intermediary directories.
FALSE	specifies not to create intermediary directories.

Examples

Example 1

```
// Declare a string variable to contain the path to the directory to
be created
string dir
dir="C:\DataQuality\my_data"

// Declare a Boolean variable for the MKDIR function call
boolean d

// Use the MKDIR function to create the C:\DataQuality\my_data
directory
d mkdir(dir)
```

Example 2

```
// Declare a string variable to contain the path to the directory to
be created
string dir
dir="C:\DataQuality\my_data"

// Declare Boolean variables for the MKDIR function call and the
optional condition
boolean b
boolean d

// Set the condition "true" to create an intermediary directory if it
does not exist
b=true

// Use the MKDIR function to create the new directory, including the
intermediary
// directory DatQuality, if it does not exist.
d mkdir(dir,b)
```

MOVEFILE Function

Moves or renames a specified file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

MOVEFILE(<old_file, new_file_name>)

Required Arguments

old_file_name
a string representing the name of the file to be moved; this can be specified as a fixed string, field name, or expression.

new_file_name
a string representing the name (including location) where the file is moved; this can be specified as a fixed string, field name, or expression.

Details

The MOVEFILE function moves a file. The directory structure must already be in place for the function to move the file to its new location. If the target file already exists, the file is not moved and false is returned.

Example

```
string old_file_name
string new_file_name
boolean rc_ok

old_file_name = "C:\mydata_copy.txt"
new_file_name = "C:\TEMP\mydata_copy.txt"

rc_ok = movefile(old_file_name, new_file_name)
```

OPEN Function

Opens a specified file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**OPEN**(<filename, openmode>)

Required Arguments

filename
a string representing the name of the file to be opened. If the file does not exist, it is created. This parameter can be specified as a fixed string, field name, or expression.

openmode
[optional] a string representing the OPENMODE to be used. This can be specified as a fixed string, field name, or expression [a = append, r = read, w = write, rw = read and write].

Details

The open method opens the file that is provided in the filename parameter. If the file does not exist and an OPENMODE is specified containing either an "a" or "w", then the file is created. If the OPENMODE is not specified, a value of false is returned.

When WRITEBYTES and WRITELINE methods write at the end of the file, unless SEEKBEGIN, SEEKCURRENT, or SEEKEND methods are used to adjust the position in the file, the information is written at the current position in the file.

If an OPENMODE of "w" is used, the WRITEBYTES and WRITELINE methods write at the current position in the file and potentially overwrite existing information in the file.

Example

```
file myfile
```

```
if ( myfile.open("data.txt") ) then ...
```

POSITION Function

Returns the current position of the cursor in a file, which is the number of bytes from the beginning of the file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is an integer representing the current position (offset) from the beginning of the file.

Syntax

<fileobject>.**POSITION**(< >)

Details

The position method returns the current position of the cursor in a file. Combined with the SEEKEND() method, it can be used to determine the size of a file.

Example

```
file f
integer byte_size

f.open("C:\filename.txt", "r")
f.seekend(0) // position cursor at end of file

// or if you want to test return codes for the method calls
// boolean rc_ok
// rc_ok = f.open("C:\filename.txt", "r")
// rc_ok = f.seekend(0) // position cursor at end of file

// The integer variable byte_size will have
// the size of the file in bytes
byte_size = f.position()

f.close()
```

READBYTES Function

Reads a certain number of bytes from a file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is the number of bytes actually read, or 0 on failure.

Syntax

<fileobject>.**READBYTES**(<number_of_bytes, buffer>)

Required Arguments

number_of_bytes
an integer specifying the number of bytes that need to be read from the file. This parameter can be specified as a number, field name, or expression.

buffer
a string that contains the bytes that are read. This parameter can be specified as a fixed string, field name, or expression.

Details

The READBYTES method reads the specified number of bytes from a file starting at the current position of the file pointer. The file pointer will be positioned after the last byte read. If the buffer is too small, only the first bytes from the file are put into the buffer.

This method is normally used to read binary files. The various format functions can be used to convert the binary information that was read.

Note that this method also reads EOL characters. When reading a windows text file like this:

```
C:\filename.txt
abc
def
```

A READBYTES(7, buffer) statement causes the field buffer to contain the following value "abc\n de". The value consists of all the information from the first line (3 bytes), followed by a CR character and an LF character (2 bytes on Windows, 1 byte on UNIX) that is represented by "\n". They are followed by the first 2 bytes from the second line. To read text files, use the READLINE() method.

Example

```
string input
file f

f.open("C:\filename.txt", "r")
f.readbytes(7, input)

// or if you want to test return codes for the method calls
// boolean rc_ok
// rc_ok = f.open("C:\filename.txt", "r")
// rc_ok = f.readbytes(7, input)
```

READLINE Function

Reads the next line from an open file.

Category:	External File
Returned data type:	Character
Note:	The returned value is a string containing the line that was read from the file.

Syntax

fileobject.**READLINE**(< >)

Details

The READLINE method reads the next line of data from an open file. A maximum of 1024 bytes are read. The text is returned. Null is returned if there was a condition such as end of file.

Example

```
file f
```

```
string input

f.open("C:\filename.txt", "r")
input=f.readline()
f.close()
```

RMDIR Function

Deletes a directory if it is empty.

Category: String

Returned data type: Boolean

Syntax

RMDIR(*directory*)

Required Argument

directory
specifies the directory to remove if it is empty.

Example

```
// Declare a string variable to contain the path to the directory to
be created
string dir
dir="C:\DataQuality\my_data"

// Declare a Boolean variable for the MKDIR function call
boolean d

// Use the MKDIR function to create the C:\DataQuality\my_data
directory
d rmdir(dir)
```

SEEKBEGIN Function

Sets the file pointer to a position starting at the beginning of the file. Returns true on success, false otherwise. The parameter specifies the position.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**SEEKBEGIN**(<position>)

Required Argument

position
an integer specifying the number of bytes that need to be moved forward from the beginning of the file. Specifying a 0 means the start of the file. This parameter can be specified as a number, field name, or expression.

Details

The SEEKBEGIN method moves the file pointer to the specified location in the file, where 0 indicates the start of the file. It returns true on success. Otherwise, false is returned. Specifying 0 means that reading starts after the first position in the file.

Example

```
file f
string input

f.open("C:\filename.txt", "r")

input = f.readline()

// return the pointer to the beginning of the file
// and read the first line again
f.seekbegin(0)
f.readline()

f.close()
```

SEEKCURRENT Function

Sets the file pointer to a position in the file relative to the current position in the file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**SEEKCURRENT**(<position>)

Required Argument

position
an integer specifying the number of bytes that need to be moved from the current position in the file. Positive values specify the number of bytes to move forward, negative values specify the number of bytes to move backward. This parameter can be specified as a number, field name, or expression.

Details

The SEEKCURRENT method moves the file pointer from the current position in the file. This method is useful when reading binary files that contain offsets to indicate where related information can be found in the file.

Example

```
file f
string input

f.open("C:\filename.txt", "r")

input = f.readline()

// The file contains 3 bytes per record followed by a CR and LF
character
// So move the pointer 3+2=5 positions back to read the beginning of
// the first line and read it again.
f.seekcurrent(-5)
f.readline()

f.close()
```

SEEKEND Function

Sets the file pointer to a position in the file counted from the end of the file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**SEEKEND**(<position>)

Required Argument

position
an integer specifying the number of bytes that need to be back from the end of the file. Specifying a 0 means the end of the file. This parameter can be specified as a number, field name, or expression.

Details

The SEEKEND method moves the file pointer backward the number of bytes that were specified, where 0 indicates the end of the file. It returns true on success otherwise false is returned.

Example

```
file f
f.open("C:\filename.txt", "rw")

// write information to the end of the file
f.seekend(0)
f.writeline("This is the end ")
f.close()
```

WRITEBYTES Function

Writes a certain number of bytes to a file.

Category: External File

Returned data type: Integer

Note: The returned value is an integer representing the number of bytes written.

Syntax

fileobject.**WRITEBYTES**(<number_of_bytes, buffer>)

Required Arguments

number_of_bytes
an integer specifying the number of bytes that is written to the file. This parameter can be specified as a number, field name, or expression.

buffer
a string that contains the bytes that need to be written. This parameter can be specified as a fixed string, field name, or expression.

Details

The WRITEBYTES method writes the specified number of bytes to a file starting at the current position in the file. This method overwrites data that exists at the current position in the file. If the current position in the file plus the number of bytes to be written is larger than the current file size, then the file size is increased.

If buffer is larger than number_of_bytes specified, then only the first number_of_bytes from buffer is written. The file needs to be opened in Write or Append mode for this method to work. The method returns the actual number of bytes written.

This method is normally used to write binary files. To write text files, the WRITELINE() method can be used.

Example

```
string input
file f

string = "this is longer than it needs to be"
f.open("C:\filename.txt", "rw")
// This will write to the beginning of the file
// Only the first 10 bytes from the string will be written
// If the file was smaller than 10 bytes it will be automatically
// appended
f.writebytes(10, input)

f.close()
```

WRITELINE Function

Writes a line to a file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**WRITELINE**(<string>)

Required Argument

string
a string specifying the information that needs to be written to the file. This parameter can be specified as a fixed string, field name, or expression.

Details

The WRITELINE() method writes the string at the current position in the file. This method overwrites data that exists at the current position in the file. If the current position in the file plus the length of the string is larger than the current file size, then the file size is increased.

The file needs to be opened in Write or Append mode for this method to work.

Example

```
file f

f.open("C:\filename.txt", "a")
f.writeline("This text will be appended to the file")

f.seekbegin(0)
f.writeline("Using seekbegin(0) and Append will
still cause the info to be written at the start of the file")

f.close()
```

16

Information/Conversion Functions

Overview

The following information and conversion functions are available for the Expression Engine Language (EEL).

Dictionary

DETERMINE_TYPE Function

Returns the type of data that the input string represents.

Category: Date

Returned data type: String

Syntax

DETERMINE_TYPE

Required Arguments

string
is the string of data.

returns
the data type the input string represents.

Details

This function analyzes a string to determine whether it is one of the following options: string, integer, Boolean, date, or real.

Examples

Example 1

```
1000
Results: integer
```

Example 2

```
1000.5
Results: real
```

ISALPHA Function

Returns a true value if the expression is a string made up entirely of alphabetic characters.

Category: Information and Conversion

Returned data type: Integer

Note: The returned value is a Boolean value, true if the "in_string" contains only alpha characters. Otherwise, the value is false.

Syntax

ISALPHA(<in_string>)

Required Argument

in string
a string of characters that is searched for any alphabetic characters.

Details

The ISALPHA function returns true if "in_string" is determined to be a string containing only alpha characters.

Examples

Example 1

```
// Expression
string letters
letters="lmnop"
string mixed
mixed="1a2b3c"

string alphatype
alphatype=isalpha(letters) // returns true
string mixedtype
mixedtype=isalpha(mixed) // returns false
```

Example 2

```
string all_Alpha
all_Alpha="abcdefghijklmnoprstuvyz"

string non_Alpha
non_Alpha="%&)#@*0123456789"
```

```
string error_message1
string error_message2

if (NOT isalpha(all_Alpha))
   error_message1 ="all_Alpha string contains alpha numeric characters"
else
   error_message1 ="all_Alpha string contains alpha numeric characters"

if(isalpha(non_Alpha))
   error_message2= "non_Alpha string contains alpha numeric characters"
else
   error_message2= "non_Alpha string does not contain alpha numeric
characters"
```

Example 3

```
string all_Alpha
string error_message
all_Alpha="abcdefghijklmnopqrstuvwxyz"
if (isalpha(all_Alpha))
   begin
        error_message= "alpha strings were identified as alpha"
   end
```

ISBLANK Function

Checks to see whether an argument contains a blank, empty value. When the argument value is blank, NULL, or contains control characters (including chr(09), chr(10), chr(11), chr(12), or chr(13)) then the function returns true. Otherwise, it returns false.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value.

Syntax

<boolean>**ISBLANK**(<argvalue>)

Required Argument

argvalue
is a string.

Details

The ISBLANK function takes the following argument types: string.

Examples

Example 1

```
string x
string y
date z
string error_message1
string error_message2

x="Hello"

if(isblank(x) )
   error_message1 = "x is blank"
else
   error_message1= "x is not blank"

if( isblank(y) )
   error_message2 =" String y value is blank"
else
   error_message2 =" String y value is not blank"
```

Example 2

```
string toCheck
boolean result
toCheck = " "

result = isblank (toCheck)
//or
result = isblank(" ")
//both return True
```

ISNULL Function

Checks if an argument value contains a null value. When the argument value is null, the function returns true. Otherwise, it returns false.

Category: Information and Conversion

Returned data type: Integer

Note: The returned value is a Boolean value.

Syntax

<boolean>**ISNULL**(<argvalue>)

Required Argument

argvalue
string, date, integer, real, Boolean.

Details

The ISNULL function takes the following argument types: string, date integer, real, Boolean.

Examples

Example 1

```
// Expression
if State <> "NC" OR isnull(State)
   return true
else
   return false
```

Example 2

```
integer x
string y
string error_message1
string error_message2

y="Hello"

if(isnull(x) )
   error_message1 = "Integer x is null"
else
   error_message1= "Integer x is not null"
```

```
if( isnull(y) )
   error_message2 =" String y value is null"
else
   error_message2 =" String y value is not null"
```

ISNUMBER Function

Checks if an argument value contains a numerical value. When the argument value is a number, the function returns true. Otherwise, it returns false.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value.

Syntax

argvaluebooleanISNUMBER()

Required Argument

argvalue
string

Details

The ISNUMBER function takes the following argument types: string.

Example

```
string x
string y
string z
string error_message1
string error_message2
string error_message3

x ="5"
y="Hello"
z="01/01/10"

if(isnumber(x) )
   error_message1 = "String x is a number"
else
```

```
   error_message1= "String x is not a number"

if( isnumber(y) )
   error_message2 =" String y value is a number"
else
   error_message2 =" String y value is not a number"

if( isnumber(z) )
   error_message3 = "String z value is a number"
else
   error_message3 = "String z value is not a number"
```

LOCALE Function

Sets the locale.

Category:	Information and Conversion
Returned data type:	String
Notes:	The returned value is a string of the current locale setting. This function affects certain operations such as converting and date operations. This function returns the previous locale. If no parameter is passed, the current locale is returned. The locale setting is a global setting.

Syntax

<string>**LOCALE**(< >)

<string>**LOCALE**(<“locale_string”>)

Details

If a parameter is specified, it is set. Otherwise, it is retrieved. If setting, the old locale is retrieved.

The following values can be set in the LOCALE() function:

- You can use a two-character abbreviation for the US and UK locales
- A three-character abbreviation can be used for some countries, like GER, FRA, or DEU

Here are some examples:

```
my_locale = locale("DEU")
my_locale = locale("German")
my_locale = locale("German_Germany")
my_locale = locale("German_Germany.1252")
```

The LOCALE() function returns the current setting using Country_Language.codepage notations.

Note: If you use ("#iso8601"), the date is set in the ISO8601 format.

Example

```
string currentSetting
string newSetting

currentSetting = locale();

newSetting = locale("FRA");
```

TOBOOLEAN Function

Converts the argument to a Boolean value.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value that is returned if the argument value can be converted to a Boolean value

Syntax

boolean**TOBOOLEAN**(value<>)

Required Argument

value
is passed in as one of the following: real, integer, string, or date.

Example

```
boolean convertedValue
integer result
result = 1
convertedValue = toboolean(result)
Print (convertedValue)
```

TYPEOF Function

Identifies the data type of the passed in value.

Category: Information and Conversion

Returned data type: Character

Note: The returned value is one of the following strings: Boolean variable return Boolean, integer variable return integer, real variable return real, string variable return string.

Syntax

<string> **TYPEOF**(in_value)

Optional Argument

in value
variable that is evaluated.

Examples

Example 1

```
// Expression
string hello
hello="hello"

boolean error
error=false

// variable that will contain the type
string type
type=typeof(hello)

// type should be string
if(type<>"string") then
   error=true
```

Example 2

```
string content
content = "Today is sunny"

hidden integer one
one =1

hidden real pi
pi=3.1415962

hidden boolean test
test=false

hidden string type

type= typeof(content);

if (type == "string")
   begin
        error_message="The data type for variable 'Content' is string"
   end

type=typeof(one)
if (type == "integer")
   begin
        error_message="The data type for variable 'one' is integer"
   end
type= typeof(pi);

if (type == "real")
   begin
        error_message="The data type for variable 'real' was real"
   end

type= typeof(test);

if (type == "boolean")
   begin
        error_message="The data type for variable 'test' was boolean"
   end
```

17

Mathematical Functions

Overview

The following mathematical functions are available for the Expression Engine Language (EEL).

Dictionary

ABS Function

Returns the absolute value of a number.

Category: Mathematical

Returned data type: Real

Note: The ABS function returns a nonnegative number that is equal in magnitude to the magnitude of the argument.

Syntax

ABS(*argument*)

Required Argument

argument
specifies a value that has a real data type; this can be specified as a numeric constant, field name, or expression.

Examples

Example 1

Statements	Results
x = abs(3.5)	// outputs 3.5
x = abs(-7)	// outputs 7
x = abs(-3*1.5)	// outputs 4.5

Example 2

```
real seconds_diff
seconds_diff = abs((date1 - date2) * 86400)
// The number 86400 represents the total number of seconds in a day
```

CEIL Function

Returns the smallest integer that is greater than or equal to the argument.

Category: Mathematical

Returned data type: Real

Syntax

CEIL(*argument*)

Required Argument

argument
specifies a value that has a real data type; this can be specified as a numeric constant, field name, or expression.

Details

This is also called rounding up (ceiling).

Example

```
x = ceil(3.5)
// outputs 4

x = ceil(-3.5)
// outputs -3

x = ceil(-3)
// outputs -3

x = ceil(-3*1.5)
// outputs -4
```

FLOOR Function

Returns the largest integer that is less than or equal to the argument.

Category:	Mathematical
Returned data type:	Real

Syntax

FLOOR(*argument*)

Required Argument

argument
specifies a value that has a data type of real; this can be specified as a numeric constant, field name, or expression.

Details

This is also called rounding down.

Example

```
x = floor(3.5)
// outputs 3

x = floor(-3.5)
// outputs -4

x = floor(-3)
// outputs -3

x = floor(-3*1.5)
// outputs -5
```

MAX Function

Returns the maximum value of a series of values.

Category:	Mathematical
Returned data type:	Real

Syntax

MAX(*argument1*< , *argument2*, ...>)

Required Argument

argument1
specifies a value that has a real data type; this can be specified as a numeric constant, field name, or expression.

Optional Argument

argument2, ...
specifies one or more values that have a real data type; these can be specified as a numeric constant, field name, or expression.

Details

The function returns NULL if all values are NULL.

Example

```
x = max(1, 3, -2)
 // outputs 3

x = max(1, null, 3)
 // outputs 3

x = max(-3)
// outputs -3

x = max(4, -3*1.5)
 // outputs 4
```

MIN Function

Returns the minimum value of a series of values.

Category: Mathematical

Returned data type: Real

Syntax

MIN(*argument1*<, *argument2*, ...>)

Required Argument

argument1
specifies a value that has a data type of real; this can be specified as a numeric constant, field name, or expression.

Optional Argument

argument2, ...
specifies one or more values that have a real data type; these can be specified as a numeric constant, field name, or expression.

Details

The function returns NULL if all values are NULL.

Example

```
x = min(1, 3, -2) // outputs -2
x = min(1, null, 3) // outputs 1
x = min(-3) // outputs -3
x = min(4, -3*1.5) // outputs -4.5
```

POW Function

Raises a number to the specified power.

Category: Mathematical

Returned data type: Real

Syntax

POW(*x*, *y*)

Required Arguments

x
specifies a value that has a data type of real and indicates the base number to raise; this can be specified as a numeric constant, field name, or expression.

y
specifies a value that has data type of real and indicates the power to raise to; this can be specified as a numeric constant, field name, or expression.

Details

The POW function raises *x* to the power *y*, x^y.

Example

Statements	Results
x = pow(5,2)	// outputs 25
x = pow(5,-2)	// outputs 0.04
x = pow(16,0.5)	// outputs 4

ROUND Function

Rounds a number to the nearest number with the specified decimal places.

Category: Mathematical

Returned data type: Real

Syntax

ROUND(<*argument*><*decimals*>)

Required Argument

argument
a value with a data type of real that can be specified as a numeric constant, field name, or expression.

Optional Argument

decimals
specifies a numeric constant, field name, or expression that indicates the number of decimal places to provide in the result of the rounding operation. A positive value for decimals is used to round to the right of the decimal point. A negative value is used to the left of the decimal point.

Default 0

Example

```
x = round(1.2345,1)
 // outputs 1.2

x = round(1.449,2)
 // outputs 1.45

x = round(9.8765,1)
 // outputs 9.9

x = round(9.8765)
 // outputs 10
```

18

Regular Expression Functions

Overview

The regular expression (regex) object enables you to perform regular expression searches of strings in Expression Engine Language (EEL).

Dictionary

COMPILE Function

Compiles a valid regular expression using the specified encoding.

Category: Regular Expression

Returned data type: Integer

Syntax

r.**COMPILE**(*regex* < ,*encoding* >)

Required Argument

regex
specifies a PCRE-compatible regular expression.

Optional Argument

encoding
specifies a string that defines the encoding constant shown Encoding. Use a value from the Encoding column.

Default The default encoding for the operating system.

Details

The compile function is used with a regular expression object. You must define a regular expression object first as a variable before you can use COMPILE( ) to compile a PCRE-compatible regular expression. Regular expressions can be used to do advanced pattern matching (and in some cases pattern replacement). Use the other functions listed below to find patterns in a given string (which can be a variable), to determine the length of matching patterns, and to replace patterns.

The function returns 1 if the regular expression compilation is successful. Otherwise, it returns 0. Failure could be due to an incorrectly formatted regular expression or possibly an invalid encoding constant.

For performance reasons, it is best to compile a regular expression in a preprocessing step in the **Expression** node. This means that the regular expression is compiled just once before data rows are processed by the **Expression** node.

Note: The sample code in this section generally places the COMPILE() function on the **Expression** tab with the rest of the expression code for clarity.

In some cases, you might need to compile the regular expression before every row is evaluated. For example, you can use a variable to define the regular expression that you want to compile. The variable might come from the data row itself, and you would need to recompile the regular expression for each row to have the pattern searching work correctly.

The syntax for PCRE-compatible regular expressions is described on the PCREPattern Man Page (https://www.pcre.org/original/doc/html/

pcrepattern.html). PCRE-compatible regular expressions do not use the "/" character as a delimiter. Internal options are enclosed between "(?" and ")" in the expression. For example, compiling the Perl expression "`/index/i`" is successful. The expression matches only the string "`/index/i`" because the / character has no special meaning. The Perl expression "`/index/i`" is expressed in PCRE-compatible format as "`(?i)index`". For additional details about the PCRE internal options, see https://www.pcre.org/original/doc/html/pcrepattern.html#SEC13.

Take care to design regular expressions that match only the patterns that you want to search for. Poorly written regular expression code can require a lot of additional processing that can negatively impact performance.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//You must define a regex object
regex r
//Then compile your regular expression
//This example will match any single digit in an input string
r.compile ("[0-9]","ISO-8859-1")
// Terminate the expression node processing
seteof()
```

See Also

- “FINDFIRST Function” on page 423
- “FINDNEXT Function” on page 425
- “MATCHLENGTH Function” on page 426
- “MATCHSTART Function” on page 428
- “REPLACE Function” on page 429
- “SUBSTRINGCOUNT Function” on page 431
- “SUBSTRINGLENGTH Function” on page 433
- “SUBSTRINGSTART Function” on page 435

FINDFIRST Function

Searches the specified string for a pattern match using an already compiled regular expression.

Category: Regular Expression

Returned data type: Boolean

Syntax

r.**FINDFIRST**(*input*)

Required Argument

input

specifies the string value in which you want to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue"). This can also be a variable already defined in your expression code or passed to the expression node as a column from a previous node (MyValue or "My Value").

Requirement *input* must not be NULL or blank.

Details

The FINDFIRST function indicates whether one or more pattern matches were found in the input. This function can be used to enter a logical loop that pulls out a series of matched patterns from an input string. The function returns TRUE if a pattern match is found. A FALSE return value indicates that no match is found and that processing can continue.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//You must define a regex object
regex r
//Then compile your regular expression. This one will match any single
// uppercase letter
r.compile("[A-Z]")
// If a pattern match is found this will evaluate to 1 (TRUE)
if r.findfirst("Abc")
// Print the output to the statistics file. You must run
// this job for stats to be written. A preview will not generate
// a message in the log.
print("Found match starting at " & r.matchstart() & " length " &
r.matchlength())
```

```
// Terminate the expression node processing
seteof()
```

FINDNEXT Function

Continues to search the string for the next match after using the FINDNEXT() function.

Category:	Regular Expression
Returned data type:	Boolean

Syntax

r.**FINDNEXT**(*input*)

Required Argument

input

specifies the string value in which you want to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue"). This can also be a variable already defined in your expression code or passed to your expression node as a column from a previous node (MyValue or "My Value").

Requirement *input* must not be NULL or blank.

Details

The FINDNEXT function indicates that another pattern match has been found after FINDFIRST() has been used. Using a "While" statement loop lets you iterate through all potential pattern matches using this function as long as the return value is equal to true.

The function returns TRUE if a pattern match was found. Otherwise, it returns FALSE.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF( ) function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows.

Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
// Define some string variables
string MyString
string MySubString

// Set one to some sample input
MyString = "DwAwTxAxFyLyUzXz"

//You must define a regex object
regex r
// Then compile your regular expression
// This one will match any single uppercase letter
r.compile("[A-Z]")
// Find the first pattern match
if r.findfirst(MyString)
   begin
   // Pull the pattern from MyString and place it into MySubString
   MySubString = mid(MyString, r.matchstart(),r.matchlength())
   // Use pushrow to create new rows - this is purely for the sake of
   // clarity in the example
   pushrow()
   // Create a while loop that continues to look for matches
   while r.findnext(MyString)
      begin
      // Pull the pattern from MyString and place it into MySubString
again
      MySubString = mid( MyString, r.matchstart(),r.matchlength())
      // Just for display again
      pushrow()
      end
   end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

MATCHLENGTH Function

Returns the length of the last pattern match found.

Category:	Regular Expression
Interaction:	This function operates on the pattern match substring found using FINDFIRST( ) or FINDNEXT( ).
Returned data type:	Integer

Syntax

r.**MATCHLENGTH**( )

Without Arguments

There is argument for this function.

Details

Use the MATCHLENGTH function to determine the length in characters of the currently matched pattern found with FINDFIRST( ) or FINDNEXT( ). Used in conjunction with MATCHSTART( ), this function can be used to find matching substrings and populate variables in your expression code.

The function returns a positive integer value that represents the number of characters found to be a pattern match of the regular expression. NULL is returned if there is no substring currently under consideration and therefore no length to return.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
// Define some variables
integer i
string MyString

//Supply some values for the variables
i = 0
MyString = "DataFlux"
// Uncomment the line below to see the value of variable i change
//MyString = "Data_Management_Studio"

//You must define a regex object
regex r
//Then compile your regular expression.
// This expression will match as many "word" characters as it can
// (alphanumerics and undescore)
r.compile("\w*")
// If a pattern match is found then set i to show the length of
// the captured substring
if r.findfirst(MyString) then i = r.matchlength()
// Terminate the expression node processing
```

```
seteof()
```

MATCHSTART Function

Returns the location of the last pattern match found.

Category: Regular Expression

Returned data type: Integer

Syntax

r.**MATCHSTART**(*input*)

Required Argument

input

specifies a string value in which you want to search for the pattern defined by your compiled regular expression.

Requirement *input* must not be NULL or blank.

Details

The MATCHSTART function returns the starting character position of a substring that has been matched to the regular expression. NULL is returned if there is no substring currently under consideration and therefore no length to return. A logical loop can be used to iterate through all matching substrings. The MATCHLENGTH( ) function can be used in conjunction with MATCHSTART( ) to pull out matching substrings so that comparisons can be made to other values or to values stored in other variables.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
// Define some string variables
```

```
string MyString
string MySubString
integer StartLocation

// Set one to some sample input
MyString = "00AA111BBB2222CCCC"
// Will hold the starting location of matched patterns
StartLocation = 0

//You must define a regex object
regex r
// Then compile your regular expression
// This one will match any single uppercase letter
r.compile("[A-Z]+")
// Find the first pattern match
if r.findfirst(MyString)
   begin
   // Pull the pattern from MyString and place it into MySubString
   MySubString = mid(MyString, r.matchstart(),r.matchlength())
   // Use pushrow to create new rows - this is purely for the sake of
   // clarity in the example
   pushrow()
   // Create a while loop that continues to look for matches
   while r.findnext(MyString)
      begin
      // Pull the pattern from MyString and place it into MySubString
      again
      MySubString = mid( MyString, r.matchstart(),r.matchlength())
      // Set StartLocation to the starting point of each pattern found
      StartLocation = r.matchstart()
      // Just for display again
      pushrow()
      end
   end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

REPLACE Function

Searches for the first string, and replaces it with the second. This differs from the REPLACE() function used outside of the regex object.

Category: Regular Expression

Returned data type: String

Syntax

r.**REPLACE**(*input-string*, *replacement–value*)

Required Arguments

input-string
specifies a string value in which you want to search for and replaced in the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue"). This can also be a variable already defined in your expression code or passed to your expression node as a column from a previous node (MyValue or "My Value").

Requirement *input-string* must not be NULL or blank.

replacement–value
specifies a string to replace *input-string* that was matched by the compiled regular expression.

Details

The REPLACE function extends the capabilities of the regex object from simply finding patterns that match a regular expression to replacing the matching substring with a new value. For example, if you wanted to match all substrings that match a pattern of two hyphens with any letter in between (-A-, -B-) and replace with a single letter (Z), you would compile your regular expression for finding the hyphen/letter/hyphen pattern. Then you would use "Z" as the replacement value of the REPLACE() function passing in a variable or string value for input.

The return value is a string value with the replacement made if a replacement could indeed be made given the regular expression in play and the value supplied for input. If no replacement could be made, then the original value for input is returned

There are limitations to this functionality. You cannot easily replace the matched substring with a "captured" part of that substring. In the earlier example, you would have to parse the matched substring after it was found using FINDFIRST( ) or FINDNEXT( ) and create the replacement value based on that operation. But matched patterns can be of variable length, so guessing the position of parts of substrings can be tricky.

Compare this to similar functionality provided with using regular expressions as part of standardization definitions. In the case of QKB definitions that use regular expressions, much smarter replacements can be made because the regular expression engine enables you to use captured substrings in replacement values.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define two string variables
string MyString
string MyNewString

// Provide a value for MyString
MyString = "12Flux"

// Defined a regular expression object variable
regex r
// Compile a regular expression that will look for a series of digits
// either 2 or 3 digits long
r. compile("\d{2,3}")
// Use the replace function to place "Data" in place of the found
// pattern and save that in a new string variable.
// If you change MyString to 1234 or 12345 you can see the
// difference in how the pattern is found
MyNewString = r.replace(MyString,"Data")
// Terminate the expression node processing
seteof()
```

SUBSTRINGCOUNT Function

Returns the number of sub-patterns found to have matched the pattern specified by the compiled regular expression.

Category:	Regular Expression
Requirement:	The regular expression must contain sub-patterns that can be used to match patterns.
Returned data type:	Integer

Syntax

r.**SUBSTRINGCOUNT**( )

Without Arguments

There is no argument for this function.

Details

Use the SUBSTRINGCOUNT function to find the total number of sub-patterns found to have matched the regular expression. Normally simple regular expressions evaluate to "1", but if you design regular expressions using sub-patterns then this function will return the number found.

The function returns a positive integer that specifies the number of substrings found to have matched the regular expression. A "0" is returned if no substrings are found.

The syntax for using subpatterns is open and closed parentheses. For example:

```
(Mr|Mrs) Smith
```

For this example, the sub-pattern is the "(Mr|Mrs)". Using this function returns the number "2" for the count of substrings since the entire string is considered the first sub-pattern. The part inside the parentheses is the second sub-pattern.

This function can provide the upper number for a logical loop using the FOR command so that your code can iterate through the matched sub-patterns for comparison to other values.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
```

```
// Find the first substring
if r.findfirst(MyString)
   begin
      // Use the "substring" functions to find the number of substrings
      SSC = r.substringcount()
      // Loop through substrings
      for i = 1 to SSC
      begin
      // Then pull out substrings
      SSS = r.substringstart(i)
      SSL = r.substringlength(i)
      MyString2 = mid(MyString,SSS,SSL)
      // Place the substrings in a data row
      pushrow()
      end
   end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

SUBSTRINGLENGTH Function

Returns the length of the n captured sub-pattern.

Category:	Regular Expression
Requirement:	The regular expression must contain sub-patterns that can be used to match patterns.
Returned data type:	Integer

Syntax

r.**SUBSTRINGLENGTH**(*n*)

Required Argument

n

specifies a positive integer that indicates the substring whose length you want to be returned.

Requirement *n* must not be NULL or blank.

Details

The SUBSTRINGLENGTH function returns a positive integer value that represents the number of characters found to be a sub-pattern match of the regular expression. NULL is returned if there is no substring currently under consideration and therefore no length to return. For more information about working with sub-patterns, see the Details section of "SUBSTRINGCOUNT Function" on page 431.

Most simple regular expressions do not have sub-patterns, and this function behaves similarly to MATCHLENGTH( ). However, if your regular expression does use sub-patterns, then this function can be used to find the length of individually captured sub-patterns found within the overall matched pattern.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
   begin
      // Use the "substring" functions to find the number of substrings
      SSC = r.substringcount()
      // Loop through substrings
      for i = 1 to SSC
      begin
```

```
        // Then pull out substrings
        SSS = r.substringstart(i)
        SSL = r.substringlength(i)
        MyString2 = mid(MyString,SSS,SSL)
        // Place the substrings in a data row
        pushrow()
        end
    end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

SUBSTRINGSTART Function

Returns the start location of the n captured sub-pattern.

Category:	Regular Expression
Requirement:	The regular expression must contain sub-patterns that can be used to match patterns.
Returned data type:	Integer

Syntax

r.**SUBSTRINGSTART**(*n*)

Required Argument

n

specifies a positive integer that indicates the sub-pattern whose starting location you want to be returned.

Requirement *n* must not be NULL or blank.

Details

The SUBSTRINGSTART function takes the input integer *n* that you supply and returns a starting location for the sub-pattern represented by that input integer. NULL is returned if there is no substring currently under consideration and therefore no location to return.

Use SUBSTRINGCOUNT( ) to determine the number of sub-patterns under consideration. Use SUBSTRINGLENGTH( ) with this function to pull out the matched sub-patterns and use them in evaluation logic of your expression code.

Most simple regular expressions will not have sub-patterns and this function will behave similarly to MATCHSTART(). However, if your regular expression does use sub-patterns, then this function can be used to find the starting point of individually captured sub-patterns found within the overall matched pattern.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
   begin
      // Use the "substring" functions to find the number of substrings
      SSC = r.substringcount()
      // Loop through substrings
      for i = 1 to SSC
      begin
      // Then pull out substrings
      SSS = r.substringstart(i)
      SSL = r.substringlength(i)
      MyString2 = mid(MyString,SSS,SSL)
      // Place the substrings in a data row
      pushrow()
      end
   end
// Terminate the expression node processing
```

```
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

19

Search Functions

Overview

The following search function is available for the Expression Engine Language (EEL).

Dictionary

INLIST Function

Returns TRUE if the target parameter matches any of the value parameters.

Category:	Search
Returned data type:	Boolean

Syntax

INLIST(*target_parameter*, *value_parameter1* <, *value_parameter2*, ...>)

Required Arguments

target_parameter
specifies a string, integer or date value to search.

value_parameter1
specifies a string, integer or date values to search for in *target_parameter*.

Optional Argument

value_parameter2, ...
specifies one or more string, integer or date values to search for in *target_parameter*.

Details

target_parameter is compared against each *value_parameter*. TRUE is returned if a match is found. Otherwise, FALSE is returned.

Example

```
string error_message

integer a
a=5
integer b
b=5

if (inlist(a,3,5)<>true)
   error_message="integer 5 not found in argument list of 3,5 "
else
   error_message="integer 5 was found in argument list of 3,5 "

print(error_message,false)
```

20

String Functions

Overview

There are several functions available in Expression Engine Language (EEL) that affect the built-in string data type.

Dictionary

APARSE Function

Parses a string into words and returns the number of words found and places the words in an array.

Category: String

Returned data type: Integer

Syntax

APARSE(*string*, *delimiter*, *word_list*)

Required Arguments

string
specifies the string that needs to be separated into words; this can be specified as fixed string, field name, or expression

Restriction *string* should not be NULL, it causes a run-time error.

Note If *string* is empty (""), a value of 1 is returned and *word_list* has one element that contains an empty string.

delimiter
specifies a string that contains the character to be used as delimiter when separating the string into words; this can be specified as fixed string, field name, or expression

Restriction If multiple characters are specified, only the last character is used.

word_list
specifies a string array that contains the words that were found during parsing; this is specified as a field name

Comparisons

The parse function is similar. It returns individual string fields instead of a string array, the string fields must be specified as part of the function invocation. The APARSE function does not have this restriction and can therefore be used when the maximum number of words is not known in advance.

Example

```
string = "one:two:three"
delimiter = ":"
nwords = aparse(string, delimiter, word_list) // outputs 3
first_word = word_list.get(1) // outputs "one"
last_word = word_list.get(nwords) // outputs "three"
```

ASC Function

Returns the position of a character in the ASCII collating sequence.

Category: String

Returned data type: Integer

Syntax

ASC(*string*)

Required Argument

string
specifies the character that needs to be found in the ASCII collating sequence; this can be specified as character constant, field name, or expression.

Restriction If multiple characters are specified, only the first character is used.

Details

See Appendix A: ASCII Values for a complete list of ASCII values.

Example

```
ascii_value = asc("a") // outputs 97
character_content = chr(97) // outputs the letter "a"
```

CHR Function

Returns an ASCII character for an ASCII code.

Category: String

Returned data type: String

Syntax

CHR(<*n*>)

Required Argument

n
specifies an integer that represents a specific ASCII character; this can be specified as a numeric constant, a field name, or an expression

Details

The CHR function returns n^{th} character in the ASCII collating sequence.

Note: The CHR function can also be used to return any Unicode character when passed a Unicode code point.

See Appendix A: ASCII Values for a complete list of ASCII values.

Examples

Example 1

```
character_content = chr(97) // outputs the letter "a"
ascii_value = asc("a") // outputs 97
```

Example 2

The following examples support Unicode code points:

```
string input_string // this is a string that could contain greek
characters
string(1) character
boolean greek_capital
for i=1 to len(input_string)
begin
   character=mid(input_string,i,1)
   if chr(character)>=913 and chr(character)<=939 then
      greek_capital=true
end
```

COMPARE Function

Returns the result of comparing two strings.

Category:	String
Returned data type:	Integer

Syntax

COMPARE(*string1*, *string2*<,*case-insensitive*>)

Required Arguments

string1
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

string2
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

Optional Argument

case-insensitive
specifies a Boolean string that indicates whether to compare case-insensitive strings; this can be specified as string constant, field name, or expression.

TRUE specifies that the string comparison is not case sensitive.

FALSE specifies that the comparison is case sensitive.

Default FALSE

Details

The MATCH_STRING function compares two strings lexicographically and can be used to do string comparisons using wildcards.

The return value is an integer representing the result of a lexicographical comparison of the two strings:

```
[-1 = string1 < string 2,
0 = string1 equals string2,
1 = string1 > string2]
```

To check whether two strings are equal, it is more efficient to use the == operator, for example:

```
if string1 == string2 then match=true
```

Example

```
// hallo comes before hello when alphabetically sorted
rc = compare("hello", "hallo" ) // outputs 1

// Hello comes before hello when alphabetically sorted
rc = compare("Hello", "hello" ) // outputs -1

modifier = null
rc = compare("Hello", "hello", modifier ) // outputs -1
rc = compare("Hello", "hello", true ) // outputs 0
```

EDIT_DISTANCE Function

Returns the number of corrections that would need to be applied to transform one string into the other.

Category: String

Returned data type: Integer

Syntax

EDIT_DISTANCE(*"string1"*, *"string2"*)

Required Arguments

"string1"
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

"string2"
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

Details

The EDIT_DISTANCE function returns the number of corrections that need to be applied to transform *"string1"* into *"string2"*.

Example

```
distance = edit_distance("hello", "hllo" )
 // outputs 1

distance = edit_distance("hello", "hlelo" )
 // outputs 2

distance = edit_distance("hello", "hey" )
 // outputs 3
```

HAS_CONTROL_CHARS Function

Determines whether a string contains ASCII control characters.

Category: String

Returned data type: Boolean

Syntax

HAS_CONTROL_CHAR(*string*)

Required Argument

string
specifies a string to be search for the existence of ASCII control characters.

Details

The HAS_CONTROL_CHARS function can be used to identify non-printable ASCII control characters as found on the ASCII character table. A return value of TRUE indicates that control characters were found. A return value of FALSE indicates that control characters were not found.

Note: The only control character the HAS_CONTROL_CHARS function does not detect is 0 (null character).

See Appendix B: ASCII Control Characters for a list of control characters.

Example

```
boolean result_1

string error_message1

string test
test="Contol character: "&chr(13)
result_1=has_control_chars(test)
if(result_1)
   error_message1 = "test string contains control character"
else
   error_message1 = "test string does not contain control character"
```

INSTR Function

Returns the position of one string within another string.

Category: String

Returned data type: Integer

Syntax

INSTR(*source*, *excerpt*<, *count*>)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

excerpt
specifies a string to search for within *source*; this can be specified as string constant, field name, or expression.

Optional Argument

count
specifies an integer that indicates the occurrence of *excerpt* to search for; this can be specified as numeric constant, field name, or expression. For example, a value of 2 indicates to search for the second occurrence of *excerpt* in *source*.

Details

The INSTR function searches *source* from left to right for the *count*-th occurrence of *excerpt*. If *excerpt* is not found in *source*, the function returns a value of 0.

Example

```
source = "This is a simple sentence."
excerpt = "is"

position = instr(source, excerpt, 1) // outputs 3
position = instr(source, excerpt, 2) // outputs 6
```

LEFT Function

Returns the left-most characters of a string.

Category:	String
Returned data type:	String

Syntax

LEFT(*source*, *count*)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

Note If source is NULL, the function returns a NULL value.

count
specifies an integer that indicates how many characters to return; this can be specified as numeric constant, field name, or expression.

Note When a count of zero or less is specified, an empty string is returned.

Example

```
source = "abcdefg"
result = left(source, 4) // outputs the string "abcd"
```

LEN Function

Returns the length of a string.

Category: String

Returned data type: Integer

Syntax

LEN(*source*)

Required Argument

source
specifies a string for which the length needs to be determined; this can be specified as string constant, field name, or expression.

Note The length of an empty string ("") is zero. If *source* is NULL, the function returns a NULL value.

Tip To remove leading and trailing blanks, use the trim function.

Examples

Example 1

```
string(30) source
source = "abcdefg"
length_string = len(source) // outputs 7

source = " abcdefg "
length_string = len(source) // outputs 11

source = " " // source contains a blank
length_string = len(source) // outputs 1
```

Example 2

```
function nvlString
return string len
//!params string inval string defaultVal

if isnull(parameter(1)) then return parameter(2)
else return parameter(1)
end function
```

LOWER Function

Converts a string to lowercase.

Category: String

Returned data type: String

Syntax

LOWER(*source*)

Required Argument

source
specifies a string; this can be specified as string constant, field name, or expression.

Note If *source* is NULL, the function returns a NULL value.

Example

```
source = "MÜNCHEN in Germany"
lowcase_string = lower(source) // outputs "münchen in germany"
```

MATCH_STRING Function

Determines whether the first string matches the second string, which can contain wildcards.

Category: String

Returned data type: Boolean

Syntax

MATCH_STRING(*string1*, *string2*)

Required Arguments

string1
specifies a string to search; this can be specified as string constant, field name, or expression.

Note If source is NULL the function returns a NULL value.

string2
specifies a string that represents a search pattern; this can be specified as string constant, field name, or expression.

Details

The MATCH_STRING function searches *string1* using the search pattern specified in *string2*. If a match was found, true is returned. Otherwise, false is returned.

Search strings can include wildcards in the leading (*ABC) and trailing (ABC*) position, or a combination of the two (*ABC*). Wildcards within a string are invalid (A*BC).

A question mark can be used as a wildcard but is matched only to a character. For example, AB? will match ABC, not AB.

To execute a search for a character that is used as a wildcard, precede the character with a backslash. This denotes that the character should be used literally and not as a wildcard. Valid search strings include: *BCD*, *B?D*, *BCDE, *BC?E, *BCD?, ABCD*, AB?D*, ?BCD*, *B??*, *B\?* (will match the literal string AB?\E). An invalid example is: AB*DE.

For more complex searches, use regular expressions instead of the MATCH_STRING() function.

Example

```
string1 = "Monday is sunny, Tuesday is rainy & Wednesday is windy"
string2 = "Tuesday is"
match = match_string(string1, string2) // outputs false
string2 = "*Tuesday is*"
match = match_string(string1, string2) // outputs true
```

MID Function

Extracts a substring from an argument.

Category: String

Returned data type: String

Syntax

MID(*source*, *position*<, *length*>)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

position
specifies an integer that is the beginning character position; this can be specified as a numeric constant, field name, or expression.

Optional Argument

length
specifies an integer that is the length of the substring to extract; this can be specified as a numeric constant, field name, or expression.

Default If length is omitted, the remainder of the string will be extracted.

Note If length is NULL, zero, or larger than the length of the expression that remains in source after position, the remainder of the expression is returned.

Example

```
source = "06MAY15"

result = mid(source, 3, 3) // outputs "MAY"
result = mid(source, 3) // outputs "MAY15"
```

OCCURS Function

Returns the number of times one string occurs within another string.

Category: String

Returned data type: Integer

Syntax

OCCURS(*string1*, *string2*)

Required Arguments

string1
specifies the string to search.

string2
represents the substring to count.

Details

The OCCURS function returns the number of times that a substring occurs within a string.

Example

```
string s
integer o
s="one:two:three"
o=occurs(s,":")
//the value o should contain 2 at this point
```

PARSE Function

Parses a string into words and returns the number of words found and the words found.

Category: String

Returned data type: Integer

Syntax

PARSE(*string*, *delimiter*, *word1*<, *word2* , ...>)

Required Arguments

string
specifies a string with delimiters that is to be separated into words; this can be specified as fixed string, field name, or expression.

Note If *string* is NULL, the function returns a NULL value. If *string* is empty ("") a value of 1 is returned.

delimiter
specifies a character that delimits the words in a string; this can be specified as fixed string, field name, or expression.

word1
specifies a string that represents the first word found; this is specified as field names.

Optional Argument

word2, ...
specifies one or more strings that represents, in order, strings that are found after the first string; these are specified as field names.

Details

The PARSE function assigns to the provided parameters the words found in *string* that are separated by a delimiter. The return values indicates the number of words found.

Comparisons

The APARSE function is similar. The APARSE function is more flexible as you do not have to know in advance the maximum number of words. It can also be used to easily determine the last word in a string.

Example

```
string = "one:two:three"
delimiter = ":"
nwords = parse(string, delimiter, word1, word2) // outputs 3
// word1 will contain the value "one"
// word2 will contain the value "two"
```

PATTERN Function

Indicates whether a string has numbers, uppercase characters, and lowercase characters.

Category: String

Returned data type: String

Syntax

PATTERN(*string*)

Required Argument

string
specifies a string that is to be evaluated for numbers, uppercase characters, and lowercase characters; this can be specified as fixed string, field name, or expression

Note If string is NULL, the function returns a NULL value. If string is empty (""), an empty value is returned.

Details

The returned string contains a 9 in place of each number, an “A” for each uppercase character, and an “a” for each lowercase character. Other characters are not replaced.

Example

```
source_string = "12/b Abc-Str."
result = pattern(source_string) // outputs "99/a Aaa-Aaa."
```

REPLACE Function

Replaces the first occurrence of one string with another string, and returns the resulting string.

Category: String

Returned data type: String

Syntax

REPLACE(*source*, *search-string*, *replace-string*<, *count*>)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

Note If source is NULL, the function returns a NULL value.

search-string
specifies the text that is to be replaced; this can be specified as string constant, field name, or expression.

replace-string
specifies a string that is to replace *search-string*; this can be specified as string constant, field name, or expression.

Optional Argument

count
specifies an integer that represents how many replacements should be made; this can be specified as numeric constant, field name, or expression.

Note If count is omitted or set to zero, all occurrences will be replaced in the string.

Example

```
source_string =
     "It's a first! This is the first time I came in first place!"
search = "first"
replace = "second"
count = 2
result = replace(source_string, search, replace, count)
// outputs "It's a second! This is the second time I came in first
place!"
```

RIGHT Function

Returns the right-most characters of a string.

Category: String

Returned data type: String

Syntax

RIGHT(*source*, *count*)

Required Arguments

source
specifies a string to be searched; this can be specified as string constant, field name, or expression.

Note If *source* is NULL, the function returns a NULL value.

count
specifies an integer that indicates how many characters to return; this can be specified as numeric constant, field name, or expression.

Note When *count* is zero or less, an empty string is returned.

Example

```
source = "abcdefg"
```

```
result = right(source, 4) // outputs the string "defg"

source = "abcdefg "
result = right(source, 2) // outputs the string "fg "
```

SORT Function

Returns a string with its characters sorted alphabetically.

Category: String

Returned data type: String

Syntax

SORT(*source* <, *ascending*, *remove_duplicates*>)

Required Argument

source
specifies a string to sort; this can be specified as string constant, field name, or expression.

Note If *source* is NULL, the function returns a NULL value.

Optional Arguments

ascending
specifies whether the text should be sorted in ascending order; this can be specified as Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE the input string is sorted in ascending order.

FALSE the input string is sorted in descending order.

Default TRUE

remove_duplicates
specifies whether duplicate characters should be removed; this can be specified as Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE duplicate characters are removed.

FALSE duplicate characters are not removed.

Default FALSE

Details

In determining the order, special characters precede initial capital letters, which precede lowercase letters.

Example

```
source_string = "A short Sentence."
ascending = true
remove_duplicates = true
result = sort(source_string, ascending, remove_duplicates)
// outputs ".AScehnorst"
```

SORT_WORDS Function

Returns a string that consists of the words that are sorted alphabetically.

Category:	String
Returned data type:	String
Note:	Special characters such as ",.!" are not treated as separation characters.

Syntax

SORT_WORDS(*source*<*ascending*>,<*remove_duplicates*>)

Required Argument

source
specifies a string to sort; the string can be specified as string constant, field name, or expression.

Note If source is NULL, the function returns a NULL value.

Optional Arguments

ascending
specifies whether the words in the input string should be sorted in ascending order; this can be specified as a Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE the input string is sorted in ascending order.

FALSE the input string is sorted in descending order.

Default TRUE

remove_duplicates
specifies whether duplicate words should be removed; this can be specified as a Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE duplicate words are removed.

FALSE duplicate words are not removed.

Default FALSE

Details

In determining the order, words with initial capital letters precede lowercase letters. Also, words with a concatenated special character are treated as a different word. For example, first and first! are two different words.

Example

```
source_string =
     "It's a first! This is the first time I came in first place!"
ascending = true
remove_duplicates = true
result = sort_words(source_string, ascending, remove_duplicates)
// outputs "I It's This a came first first! in is place! the time"
```

TODATE Function

Converts the argument to a date value based on the locale settings on your system.

Category: String

Interaction: The TODATE( ) function depends on the default **Short date** regional setting on your Microsoft Windows environment. You can change the locale setting on Windows.

Returned data type: Date

Syntax

TODATE(*any*)

Required Argument

any
specifies a value to convert to a date.

Details

When the TODATE( ) function is evaluated, the value of the Windows **Short date** locale field is examined to determine whether the format is MM/DD/YYYY or DD/MM/YYYY. The order of the month and date is determined by the **Short date** value.

Example

```
// Declare the date variable to contain the date value
date dateval

// Use the TODATE function to populate the date variable
dateval=todate(3750)

// Returns the value:
4/7/10 12:00:00 AM
```

TOINTEGER Function

Converts the argument to an integer value.

Category: String

Returned data type: Integer

Syntax

TOINTEGER(*any*)

Required Argument

any
specifies a value to convert to an integer.

Examples

Example 1

```
if tointeger(counter)<= 5
setoutputslot(0)
else
setoutputslot(1)
```

Example 2

```
// Declare an integer variable to contain the integer value
integer intval

// Use the TOINTEGER function to populate the integer variable
intval=tointeger(3750.12345)

// Returns the value:
3750
```

See Also

For rules and special considerations when you coerce types, see "Coercion" in *Expression Language: Reference Guide*.

TOREAL Function

Converts the argument to a real value.

Category:	String
Returned data type:	Real

Syntax

TOREAL(*any*)

Required Argument

any
specifies the value that is to be converted to a real value.

Example

```
// Declare a real variable to contain the real value
real realval

// Use the TOREAL function to populate the real variable
realval=toreal(3750.12345)

// Returns the value:
3750.12345
```

See Also

For rules and special considerations when you coerce types, see "Coercion" in *Expression Language: Reference Guide*.

TOSTRING Function

Converts the argument to a string value.

Category:	String
Returned data type:	String

Syntax

TOSTRING(*any*)

Required Argument

any
specifies any non-string value to conversion to a string.

Example

```
// Declare a string variable to contain the string
String stringval

// Use the TOINTEGER function to populate the integer variable
stringval=tostring(3750.12345)

// Returns the string
3750.12345
```

See Also

For rules and special considerations when you coerce types, see "Coercion" in *Expression Language: Reference Guide*.

TRIM Function

Removes leading and trailing white space.

Category: String

Returned data type: String

Syntax

TRIM(*source*)

Required Argument

source
a string from which the leading and trailing white space needs to be removed; this can be specified as string constant, s field name, or an expression.

Note If source is NULL, the function returns a NULL value. If source is an empty value (""), the function returns an empty value.

Example

```
source = " abcd " // 2 leading and 2 trailing spaces
result = trim(source) // outputs "abcd"
length = len(source) // outputs 8
length = len(result) // outputs 4
```

UPPER Function

Converts a string to uppercase.

Category: String

Returned data type: String

Syntax

UPPER(*source*)

Required Argument

source
specifies a string constant, a field name, or an expression.

Note If *source* is NULL, the function returns a NULL value.

Example

```
source = "MÜNCHEN in Germany"
upcase_string = upper(source) // outputs "MÜNCHEN IN GERMANY"
```

USERNAME Function

In SAS Data Management Studio, the USERNAME function returns the user name of the user that is logged in to the operating system. However, in DataFlux Data Management Server the USERNAME function returns the account name for the Data Management Server process.

Category: String

Returned data type: String

Syntax

USERNAME ( )

Without Arguments

This function does not take arguments.

Example

```
// Declare the string value for the function
string user

// For Data Management Studio, use the USERNAME
// function to get the operating system user name
user = username()
```

```
// For Data Management Server, use the USERNAME
// function to get the account name for the Data
// Management Server process.
user = username()
```

21

Functional Window and Notification Window Functions

Functions for Event Stream Processing

ARRAYITEM

Returns indexed values of a referenced array.

arrayItem($*arrayfield*,*index*)

For example, suppose a Source window contains the following field in its input schema:

```
<field name="scores" type="array(dbl)"/>
```

In an event loop, you can reference the array and obtain indexed values from it as follows:

```
<function name="score">arrayItem($scores,object_id)</function>
```

Here, `object_id` is a field in the schema of the Functional window.

CONTQUERYNAME

Returns the name of the continuous query that contains the current event.

contqueryName()

```
contqueryName()=cq_1
contqueryName()=myquery
```

ENGINEMETADATA

Return the value of the engine metadata item specified by the argument.

engineMetadata(*argument*)

Table 21.1 *Arguments for ENGINEMETADATA*

Argument	Description
argument	Specifies a string whose value is an engine metadata item (for example, **`version`** or **`model`**).

```
engineMetadata('version')=1.2
engineMetadata('model')=prodmodel
```

ESPCONFIGVALUE

Return the value of the configuration value specified by the argument.

espConfigValue(*argument*)

Table 21.2 *Arguments to ESPCONFIGVALUE*

Argument	Description
argument	Specifies a configuration value. For more information about configuration values, see "Configuring the ESP

Argument	Description
	Server in an Edge Environment" in *SAS Event Stream Processing: Using the ESP Server in an Edge Environment*.

When the following value is specified in **esp-properties.yaml**:

```
modelvalues:
    magicnumber: 11
```

The function returns the following:

```
espConfigValue('meta.meteringhost')=espsrv01
product(espConfigValue('modelvalues.magicnumber'),10)=110
```

EVENTCOUNTER

Return the 0-based number of events generated by a functional window.

eventCounter()

```
string($id,'-',eventCounter())=eventid-0
string($id,'-',eventCounter())=eventid-1
string($id,'-',eventCounter())=eventid-2
```

EVENTNUMBER

Returns the 0–based number of events that are generated by the current incoming event. The number is incremented each time that the function is invoked.

eventNumber()

```
string($id,'-',eventNumber())=eventid-0
string($id,'-',eventNumber())=eventid-1
string($id,'-',eventNumber())=eventid-2
```

EVENTSPROCESSED

Returns the total number of events processed by the output window.

eventsProcessed()

```
eventsProcessed()=10000
```

EVENTTIMESTAMP

Returns the timestamp of the current event.

eventTimestamp()

```
eventTimestamp()=1506347669000000
```

INPUT

Returns the name of the event stream processing input window.

input()

```
input()=sourceWindow
```

ISFAILOVERSTANDBY

Returns **`true`** when the set of running projects contains one or more projects that are running a subscriber connector currently in hot failover standby mode. Returns **`false`** when no subscriber connectors are running in hot failover standby mode.

isFailoverStandby()

```
isFailoverStandby()=true
```

ISLASTEVENTINBLOCK

Returns true when the event being processed is the last event in the event block. Otherwise, return false.

isLastEventInBlock()

```
isLastEventInBlock()=true
```

ISNOTRETENTION

Returns true when this event is not generated by a retention policy. Returns false when this event is generated by a retention policy.

isNotRetention()

```
isNotRetention()=true
```

ISRETENTION

Returns true when the event is generated by a retention policy. Returns false when the event is not generated by a retention policy.

isRetention()

```
isRetention()=true
```

OPCODE

Returns the opcode of the current event.

opcode()

```
opcode()=insert
opcode()=upsert
opcode()=delete
```

OUTPUT

Returns the name of the event stream processing output window.

output()

```
output()=myFunctionalWindow
```

PROJECTMETADATA

Return the value of the project metadata item specified by the argument.

projectMetadata(*projectname*,*argument*)

Table 21.3 *Arguments for PROJECTMETADATA*

Argument	Description
projectname	Specifies the name of the project for which metadata is to be returned.
argument	Specifies a string whose value is a project metadata item.

The following example returns the ID of the MAS module used to score data in the project named ScoreProj.

```
projectMetadata('scoreproj','mm_linked_module1')=1.2
```

PROJECTNAME

Returns the name of the project containing the current event.

projectName()

```
projectName()=project_1
projectName()=myproject
```

TOLOWERUTF8

Converts the value of the argument to lowercase. Arguments can accommodate multibyte data (characters that require more than one byte to represent).

toLowerUtf8(*string*)

Table 21.4 *Argument for TOLOWERUTF8*

Argument	Description
string	Specifies a string.

Although the function accepts multibyte data, it does not actually process this type of data. In the following example, while the uppercase single-byte characters are converted as expected, the multibyte characters remain unchanged.

```
toLowerUtf8("ZAŻÓŁĆ GĘŚLĄ JAŹŃ")=zaŻÓŁĆ gĘŚlĄ jaŹŃ
```

TOUPPERUTF8

Converts the value of the argument to uppercase. Arguments can accommodate multibyte data.

toUpperUtf8(*string*)

Table 21.5 *Argument for TOUPPERUTF8*

Argument	Description
string	Specifies a string.

Although the function accepts multibyte data, it does not actually process this type of data. In the following example, while the lowercase single-byte characters are converted as expected, the multibyte characters remain unchanged.

```
toUpperUtf8("zażółć gęślą jaźń")=ZAżółć GęśLą JAźń
```

General Functions

ABS

Returns the absolute floating-point value of the supplied argument.

abs(*argument*)

Table 21.6 *Argument for ABS*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks. ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
abs(-55)=55
abs(44)=44
```

AND

When both arguments are true, returns true. Otherwise, returns false.

and(*argument1* , *argument2*)

Table 21.7 Arguments for AND

Argument	Description
argument1 , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks. ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

If the argument returns a numeric value, then the function returns true when the numeric value is nonzero. Otherwise, it returns false.

If the argument returns a string value:

- When the string value is 'true', the function returns true.
- When the string value is 'false', the function returns false.
- Else, when the length of the string is > 0, the function returns true, otherwise false.

```
and(gt(3,2),gt(2,5))=0
and(gt(3,2),gt(12,5))=1
and(0,1)=0
and('non empty string',55)=1
and('true',55)=1
and('',4)=0
```

BASE64DECODE

Decodes the supplied base64-encoded string.

base64Decode(*string*)

Table 21.8 Arguments for BASE64DECODE

Argument	Description
string	Specifies a base64–encoded string. There is no length limit to this string.

```
base64Decode('dGhpcyBpcyBhIHRlc3Q=')=this is a test
```

BASE64DECODEBINARY

Decodes the supplied base64-encoded string and sets the result to the binary representation of that data.

base64DecodeBinary(*string*)

Table 21.9 *Arguments for BASE64DECODEBINARY*

Argument	Description
string	Specifies a base64–encoded string. There is no length limit to this string.

```
base64DecodeBinary('dGhpcyBpcyBhIHRlc3Q=')=<binary data>
```

BASE64ENCODE

Encodes the supplied string.

base64Encode(*string*)

Table 21.10 *Argument for BASE64ENCODE*

Argument	Description
string	Specifies a text string. There is no length limit to this string.

```
base64Encode('this is a test')=dGhpcyBpcyBhIHRlc3Q=
```

BASE64ENCODEBINARY

Encodes the supplied binary data and returns an encoded string

base64Encode(binary_data)

```
base64EncodeBinary(<binary data>)=dGhpcyBpcyBhIHRlc3Q=
```

BETWEEN

When the first argument is greater than the second and less than the third, returns true. Otherwise, returns false.

Table 21.11 *Arguments for BETWEEN*

Argument	Description
argument1 , *argument2* , *argument3*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks. ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
between(20,17,30)=1
between(20,17,15)=0
```

BOOLEAN

When the supplied argument is a string, returns true when the string has length greater than 0. When the supplied argument is numeric, returns true when value is not equal to 0. When the supplied argument is a Boolean expression, returns true when the value is true. Otherwise, returns false.

boolean(*argument*)

Table 21.12 *Arguments to BOOLEAN*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character.

Argument	Description
	■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

Note: The special string values **`'true'`** and **`'false'`** are handled outside the string length > 0. If **`'true'`** , the function returns 1. If **`'false'`** , the function returns 0.

```
boolean('my string')=1
boolean('')=0
boolean(10)=1
boolean(0)=0
boolean(gt(4,7))=0
boolean(gt(7,5))=1
```

CEILING

Returns the integer value above the numeric value of the supplied argument.

ceiling(*argument*)

Table 21.13 *Argument for CEILING*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
ceiling(product(4,4.1))=17
```

COMPARE

Compares the first argument to the second. If the first argument is less than the second, then it returns -1. If the first is greater than the second, then it returns 1. If the first is equal to the second, then it returns 0.

compare(*argument1* , *argument2*)

Table 21.14 *Arguments to COMPARE*

Argument	Description
argument1 , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The type of the first argument determines whether equality is determined by a string or numeric comparison.

```
compare('bears','lions')=-1
compare('lions','bears')=1
compare('bears','bears')=0
compare(10,20)=-1
compare(20,10)=1
compare(10,10)=0
```

CONCAT

Returns a string that is the concatenation of the string values of the supplied arguments.

concat(*argument1* , *argument2* ,...<*argumentN* >)

Table 21.15 *Arguments to CONCAT*

Argument	Description
argument1 , ...*argumentN*	Specifies one of the following:

Argument	Description
	■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

A minimum of two arguments is required.

```
concat('Name: ','Joe',', Age: ',floor(sum(25,10)),'.') = Name: Joe,
Age: 35.
```

CONCATDELIM

Returns a string that is the concatenation of the supplied values separated by the specified delimiter.

concatDelim(*'delimiter'*,*argument1* , *argument2* , ...<*argumentN* >)

Table 21.16 *Arguments to CONCATDELIM*

Argument	Description
'delimiter'	Specifies a character used as a delimiter.
argument1 , ...*argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
concatDelim('.','www','sas','com') = www.sas.com
```

CONTAINS

When the string value of the first argument contains the string value of the second, it returns true. Otherwise, it returns false.

contains(*argument1* , *argument2*)

Table 21.17 *Arguments to CONTAINS*

Argument	Description
argument1 , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
contains('www.sas.com','sas') = true
contains('www.google.com','sas') = false
```

DECREMENT

Returns the numeric value of the supplied argument minus 1. This function supports only integers.

decrement(*argument*)

Table 21.18 *Arguments to DECREMENT*

Argument	Description
argument	Specifies an integer.

```
decrement(10)=9
```

DIFF

Returns the value of the first argument minus the second.

diff(*argument1* , *argument2*)

Table 21.19 *Arguments to DIFF*

Argument	Description
argument1 , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
diff(sum(8,7),22)=-7.0
```

EQUALS

Returns true when the first argument is equal to the second. Otherwise, it returns false.

equals(*argument1* , *argument2*)

Table 21.20 *Arguments to EQUALS*

Argument	Description
argument1 , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**.

Argument	Description
	■ a function.

The type of the comparison depends on the type of the first argument.

```
equals('sas.com',string('sas','.com'))=1
equals('sas.com','google.com')=0
equals(10,sum(5,5))=1
```

FALSE

Returns true when the Boolean value of the argument is false. Otherwise, it returns true.

Table 21.21 *Arguments to FALSE*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
false(0)=1
false(equals(10,10))=0
```

FLOOR

Returns the integer value below the numeric value of the supplied argument.

floor(*argument*)

Table 21.22 *Arguments to FLOOR*

Argument	Description
argument	Specifies one of the following:

Argument	Description
	■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
floor(product(3.5,7))=24
```

GT

Returns true when the first argument is greater than the second. Otherwise, it returns false.

gt(*argument* , *argument2*)

Table 21.23 *Arguments to GT*

Argument	Description
argument , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The type of the comparison depends on the type of the first argument.

```
gt(sum(10,4),13)=true
gt('internet explorer','internet explorer')=false
gt('internet explorer','netscape')=false
```

GTE

Returns true when the first argument is greater than or equal to the second. Otherwise, it returns false.

gte(*argument* , *argument2*)

Table 21.24 *Arguments for GTE*

Argument	Description
argument , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The type of the comparison depends on the type of the first argument.

```
gte(sum(10,4),13)=true
gte('internet explorer','internet explorer')=true
gte('netscape','internet explorer')=true
```

GUID

Returns a globally unique identifier.

guid()

```
guid()=46ca7b9e-b11d-41be-a3eb-be8bc8553aed
guid()=319cd2a6-1b30-4c1b-8ac7-55e9465ea066
```

INCREMENT

Returns the numeric value of the first argument + 1. This function supports only integers.

increment(*argument*)

Table 21.25 Arguments to INCREMENT

Argument	Description
argument	Specifies an integer value or a function that returns an integer value.

```
increment(10)=11
```

IF

When the Boolean value of the first argument is true, returns the second argument. Otherwise, it returns the third argument if specified.

if(*argument1* , *argument2* , <*argument3* >)

Table 21.26 Arguments for IF

Argument	Description
argument1	Specifies a Boolean expression.
argument2	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.
argument3	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
if(equals('x','x'),'one','two')=one
if(equals('x','y'),'one','two')=two
```

```
if(equals('x','y'),'one')=
```

IFNEXT

Evaluates the first argument in a pair. When the argument evaluates to true, the function returns the value of the second argument in the pair.

ifNext(*argument* , *argument2* , ...<*argumentN* >, <*argumentN+1* >)

Table 21.27 *Arguments to IFNEXT*

Argument	Description
argument	Specifies a Boolean expression.
argument2	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
ifNext(gt(20,10),'value 1',lt(20,10),'value 2')=value 1
ifNext(gt(20,100),'value 1',lt(20,100),'value 2')=value 2
```

IN

Returns true when expr0 matches expr1, or any expression between expr1 and exprN, inclusive. Otherwise, returns false.

in(*expr0* , *expr1* <, ...*exprN* >)

Table 21.28 *Arguments to IN*

Argument	Description
expr0 , *expr1*	Specifies the expressions to be evaluated. The minimum number of expressions is 2.
exprN	Specifies additional expressions to be evaluated.

INDEX

Returns the value of argumentN, where N is the numeric value of specified index.

index(*index* , *argument0* , ...<*argumentN* >)

Table 21.29 *Arguments to INDEX*

Argument	Description
index	Specifies an integer or a function that returns an integer.
argument0 , *argument1* , ...*argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The minimum number of arguments is 2.

```
index(1,'larry','moe','curly')=moe
index(random(0,4),10,20,30,40,50)=40
index(random(0,4),10,20,30,40,50)=20
```

INDEXOF

Returns the 0–based index of the string value of the first argument in the string value of the second argument. Returns -1 when the value is not found.

indexOf(*argument1* , *argument2*)

Table 21.30 *Arguments of INDEXOF*

Argument	Description
argument1 , *argument2*	Specifies a string.

```
indexOf('SAS Event Stream Processing','Stream')=10
indexOf('SAS Event Stream Processing','Google')=-1
```

INTEGER

Returns the integer value of the argument.

integer(*argument*)

Table 21.31 *Arguments to INTEGER*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
integer('88.45')=88
integer(111.23)=111
```

INTERVAL

Takes the numeric value of the first argument and compares it to the numeric values of all remaining arguments. If the numeric value of the first argument is less than one of the arguments that follow, then the value of the argument that follows that one is returned.

interval(*argument* , *argument2* , *argument3* , ...<*argumentN* >)

Table 21.32 *Arguments to INTERVAL*

Argument	Description
arguments	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character.

Argument	Description
	■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
interval(85,60,'F',70,'D',80,'C',90,'B','A')=B
interval(90,60,'F',70,'D',80,'C',90,'B','A')=A
```

ISNULL

Returns true when the argument is not set, otherwise it returns false.

isNull(*argument*)

Table 21.33 *Arguments to ISNULL*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
isNull($unresolved)=1
isNull('my string')=0
```

ISSET

Returns true when the argument is set, otherwise it returns false.

isSet(*argument*)

Table 21.34 Arguments for ISSET

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
isSet($unresolved)=0
isSet('my string')=1
```

JSON

Parses the JSON object specified in the first argument and returns a value as a function of the second argument.

json(*argument* , *argument2*)

Table 21.35 Arguments to JSON

Argument	Description
argument	Specifies a JSON object.
argument2	Specifies an evaluation string. In a name value pair, specify the name of the object whose value you want to return.

```
json('{first:"john",last:"smith",hobbies:
["running","reading","golf"]}',
             'first')=john
json('{first:"john",last:"smith",hobbies:
["running","reading","golf"]}',
     'last')=smith
json('{first:"john",last:"smith",hobbies:
["running","reading","golf"]}',
     'hobbies[1]')=reading
json(#myJson,'hobbies[1]')=reading
```

The `json` function implements most of the syntax of the JSONPath query language. For more information, see the JSONPath Syntax documentation. However, using an

underlying scripting language is not supported in the `json` function. In JSONPath, expressions that are placed in parentheses and written in a scripting language can be used as an alternative to explicit names or indexes.

The following example shows how you can use JSONPath filters to select specific elements that meet a condition. This example filters the `people` array to find objects where `age > 30` and returns the `name` property.

```
json('{
  "people": [
    {"name": "John", "age": 30},
    {"name": "Jane", "age": 25},
    {"name": "Doe", "age": 40}
  ]
}', 'people[?(@.age > 30)].name') = Doe
```

The following example shows how to access nested properties and use slicing (indexing). `people[0].name` retrieves the `name` of the first person in the `people` array. `data.numbers[1:3]` retrieves a slice of the array, specifically the second and third elements.

```
json('{
  "people": [
    {"name": "John", "age": 30},
    {"name": "Jane", "age": 25},
    {"name": "Doe", "age": 40}
  ]
}', 'people[0].name') = John

json('{
  "data": {
    "numbers": [10, 20, 30, 40, 50]
  }
}', 'data.numbers[1:3]') = [20,30]
```

The following example combines filters and nested access. This example filters the `products` array for objects with `price < 600` and retrieves their `name`.

```
json('{
  "products": [
    {"id": 101, "name": "Laptop", "price": 800},
    {"id": 102, "name": "Tablet", "price": 200},
    {"id": 103, "name": "Phone", "price": 500}
  ]
}', 'products[?(@.price < 600)].name') = ["Tablet","Phone"]
```

LASTINDEXOF

Returns the last index of the string value of the second argument in the string value of the first, or -1 when the value is not found.

lastIndexOf(*argument* , *argument2*)

Table 21.36 Arguments to LASTINDEXOF

Argument	Description
argument , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
lastIndexOf('http://www.sas.com/products/webanalytics','/')=27
```

LISTITEM

This function has two uses. 1) Parses the first argument using the specified delimiter and then returns the value at the specified index. 2) References an existing list in the specified function context and returns its value at the specified index.

listItem(*argument* , *delimiter* , *index*)

listItem(*reference* , *index*)

Table 21.37 Arguments to LISTITEM

Arguments	Description
argument	Specifies a delimited string.
delimiter	Specifies a character used as a delimiter.
index	Specifies a numeric value used as an index.
reference	Specifies a reference to a function context.

```
listItem('one,two,three,four',',',2)=three
listItem(#myList,0)=one
```

LISTSIZE

This function has two uses. 1) Parses the first argument using the specified delimiter and then returns its size. 2) References an existing list in the specified function context and returns its size.

listSize(*argument* , *delimiter*)

listSize(*reference*)

Table 21.38 *Arguments to LISTSIZE*

Argument	Description
argument	Specifies a delimited string.
delimiter	Specifies a character used as a delimiter.
reference	Specifies a reference to a function context.

```
listSize('one,two,three,four',',')=4
listSize(#myList)=4
```

LONG

Returns the long value of the argument.

long(*argument*)

Table 21.39 *Arguments to LONG*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
long('88.45')=88
long(111.23)=111
```

LT

Returns true when the first argument is less than the second. Otherwise, returns false.

lt(*argument* , *argument2*)

Table 21.40 *Arguments to LT*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.
argument2	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
lt(sum(10,4),13)=false
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

LTE

Returns true when the first argument is less than or equal to the second. Otherwise, returns false.

lte(*argument* , *argument2*)

Table 21.41 *Arguments to LTE*

Argument	Description
argument , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
lte(sum(10,3),13)=true
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

MAPVALUE

This function has two uses. 1) Parses name-value pairs from the first argument using the specified outer delimiter and the specified inner delimiter, and then extracts the value for the specified name. 2) References an existing value map in the referenced function context, and then extracts the value for the name.

mapValues(*argument* , *outerdelimiter* , *innerdelimiter* , *delimiter* , *name*)

mapValues(*#reference* , *name*)

Table 21.42 *Arguments to MAPVALUE*

Argument	Description
argument	Specifies a delimited string of name-value pairs.
outerdelimiter , *innerdelimiter delimiter*	Specifies characters used as delimiters.
name	Specifies the name in the name-value pair specified in *argument* .
#reference	Specifies a reference to a function context.

```
mapValue('first:John;last:Doe;occupation:plumber',';',':','occupation'
)=plumber
```

```
mapValue(#myMap,'occupation')=plumber
```

MAPVALUES

This function has two uses. 1) Parses name-value pairs from the first argument using the specified outer delimiter and the specified inner delimiter, and then extracts the values for each specified name. 2) References an existing value map in the referenced function context and extracts the values for each specified name.

mapValues(*argument* , *outerdelimiter* , *innerdelimiter* , *delimiter* , *name1* ,...<*nameN* >)

mapValues(*#reference* , *name1* , ...<*nameN* >)

Table 21.43 *Arguments to MAPVALUES*

Argument	Description
argument	Specifies a delimited string of name-value pairs.
outerdelimiter , *innerdelimiter delimiter*	Specifies characters used as delimiters.
name1 , ...*nameN*	Specifies the name in the name-value pair specified in *argument* .
#reference	Specifies a reference to a function context.

```
mapValues('first=John,last=Doe',',','=',':','first','last')=John:Doe
mapValues(#myMap,'first','last')=John:Doe
```

MAX

Returns the largest numeric value of all specified arguments.

max(*argument* , *argument2* , ...<*argumentN* >)

Table 21.44 *Arguments for MAX*

Argument	Description
argument , *argument2* , ...*argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character.

Argument	Description
	■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The minimum number of arguments is 1.

```
max(33,44.2,sum(1,3,2,12),-33.21)=44.2
```

MEAN

Returns the mean value of all specified arguments.

mean(*argument* , *argument2* , ...<*argumentN* >)

Table 21.45 *Arguments to MEAN*

Argument	Description
argument , *argument2* , ...*argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
mean(33,44.2,sum(1,3,2,12),-33.21)=15.4975
```

MIN

Returns the smallest numeric value of all specified arguments.

min(*argument* , *argument2* , ...<*argumentN* >)

Table 21.46 Arguments to MIN

Argument	Description
argument , *argument2* , ... *argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The minimum number of arguments is 1.

```
min(33,44.2,sum(1,3,2,12),-33.21)=-33.21
```

MOD

Returns the remainder of the first argument divided by the second.

mod(*argument* , *argument2*)

Table 21.47 Arguments to MOD

Argument	Description
argument , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
mod(10,3)=1.0
```

NEG

Returns the negative numeric value of the specified argument.

neg(*argument*)

Table 21.48 *Arguments to NEG*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
neg(55)=-55
```

NORMALIZESPACE

Returns a string that is created by replacing any extra white space in the specified argument with a single space.

normalizeSpace(*argument*)

Table 21.49 *Arguments for NORMALIZESPACE*

Argument	Description
argument	Specifies a string.

```
normalizeSpace('Sentence   with   many  spaces')=Sentence with many
spaces
```

NEQUALS

Returns true when the first argument is not equal to the second. Otherwise, returns false.

nequals(*argument* , *argument2*)

Table 21.50 *Arguments for NEQUALS*

Argument	Description
argument , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
nequals('sas.com',string('sas','.com'))=0
nequals('sas.com','google.com')=1
nequals(10,sum(5,5))=0
```

NEWLINE

Returns the number of newline characters specified. With no argument, returns a single new line.

newline(<*number*>)

Table 21.51 *Arguments to NEWLINE*

Argument	Description
number	Return *number* newline characters .

NOT

Returns true when the Boolean value of the argument is false. Otherwise, returns false.

not(*argument*)

Table 21.52 *Arguments to NOT*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
not(0)=1
not(equals(10,10))=0
```

NUMBER

Returns the numeric value of the argument.

number(*argument*)

Table 21.53 *Arguments to NUMBER*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**.

Argument	Description
	■ a function.

```
number('88.45')=88.45
number(111.23)=111.23
number(gt(2,1))=1
```

OR

Returns true when any of the supplied arguments are true. Otherwise, returns false.

or(*argument* , *argument2* , ...<*argumentN* >)

Table 21.54 *Arguments to OR*

Argument	Description
argument , *argument2* , ...*argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

The minimum number of arguments is 1.

```
or(equals('a','b'),nequals('a','b'))=1
or(equals('a','b'),nequals('a','a'))=0
```

OUTSTR

When the argument contains a string or one of a group of strings, returns the associated value. When there are no matches, returns a specified default value.

outstr(*argument* , *string1* , *value_associated_with_string1* , ...<*stringN* >, <*value_associated_with_stringN* >, *default*)

Table 21.55 Arguments to OUTSTR

Argument	Description
argument	Specifies a string.
string1 ...stringN	Specifies a string or group of strings.
value_associated_with_string1 ...value_associated_with_stringN	Specifies a value associated with a string or values associated with a group of strings.
default	Specifies a string or group of strings.

```
outstr('government spending','govern','Government','Other')=Government
outstr('spending',('govern','spend'),
       'Government or Spending','Other')=Government or Spending
outstr('bob',('govern','spend'),'Government or Spending',
      ('john','jack','bob'),'Names','Other')=Names
outstr('stream processing',('govern','spend'),'Government or Spending',
       ('john','jack','bob'),'Names','Other')=Other
```

PRECISION

Sets the decimal point precision of the first argument to the second argument.

precision(*argument* , *argument2*)

Table 21.56 Arguments to PRECISION

Argument	Description
argument , *argument2*	Specifies a numeric value.

```
precision(123.44567,2)=123.45
```

PRODUCT

Returns the product of the supplied arguments.

product(*argument* , *argument2* ...<*argumentN* >)

Table 21.57 Arguments to PRODUCT

Argument	Description
argument , *argument2* , ...*argumentN*	Specifies a numeric value or a function that returns a numeric value.

```
product(3,sum(2,4),2)=36
```

QUOTIENT

Returns the quotient of the supplied arguments.

quotient(*argument* , *argument2* ...<*argumentN* >)

Table 21.58 Arguments to QUOTIENT

Argument	Description
argument , *argument2* , ...*argumentN*	Specifies a numeric value or a function that returns a numeric value.

```
quotient(3,sum(2,4),2)=0.25
```

RANDOM

Returns a random number between the first argument and the second.

random(*argument* , *argument2*)

Table 21.59 Arguments to RANDOM

Argument	Description
argument , *argument2*	Specifies a numeric value.

```
random(100,1000)=741
random(100,1000)=356
random(100,1000)=452
precision(random(0,.5),2)=0.17
```

RGX

Runs the specified regular expression on a supplied string and returns the result. If a group is specified, the result is the content of the specified numeric regular expression group.

rgx(*regular_expression* , *string* ...<*group* >)

Table 21.60 *Arguments to RGX*

Argument	Description
regular_expression	Specifies a regular expression or a reference to a regular expression in the function context.
string	Specifies a string.
group	Specifies a numeric reference to the regular expression.

```
rgx('.*/view/([0-9]*)/([0-9]*)',
           'http://cistore-dev.unx.sas.com/products/view/23/4',
           1)=23
rgx(#myExpr,
    'http://cistore-dev.unx.sas.com/products/view/23/4'
    ,2)=4
```

RGXINDEX

Runs the specified regular expression on a supplied string. When a match is found, returns the index of the match. When no match is found, returns -1.

rgxIndex(*regular_expression* , *string* ...<*stringN* >)

Table 21.61 *Arguments to RGXINDEX*

Argument	Description
regular_expression	Specifies a regular expression or a reference to a regular expression in the function context.
string ...stringN	Specifies a string.

```
rgxIndex('developer','larry - manager','moe - tester','curly -
developer')=2
```

RGXLASTTOKEN

Uses the regular expression in the first argument as a delimiter within the regular expression of the second to find all strings separated by that expression.

rgxLastToken(*regular_expression1* , *regular_expression2* ...<*index_value* >)

Table 21.62 *Arguments to RGXLASTTOKEN*

Argument	Description
regular_expression1	Specifies a regular expression or a reference to a regular expression in the function context.
regular_expression2	Specifies a regular expression.
index_value	Specifies an index value (defaults to 0) that counts from the last token in the expression. When this value is greater than 0 and less than or equal to the number of tokens in the regular expression, the token at the value is returned. Otherwise, null is returned.

```
rgxLastToken('/','data/opt/sas/dataflux')=dataflux
rgxLastToken('/','data/opt/sas/dataflux',2)=opt
rgxLastToken('\.','www.sas.com')=com
```

RGXMATCH

Compares the regular expression in the first argument to the second argument and returns a Boolean value that indicates whether a match is found.

rgxMatch(*regular_expression1* , *string* ...<*group* >)

Table 21.63 *Arguments to RGXMATCH*

Argument	Description
regular_expression1	Specifies a regular expression or a reference to a regular expression in the function context.
string	Specifies a string.

Argument	Description
group	Specifies a numeric reference to the regular expression. When specified, the result is the content of the specified numeric regular expression group.

```
rgxMatch('(google|yahoo|bing)','http://www.google.com')=1
rgxMatch('(google|yahoo|bing)','http://www.sas.com')=0
```

RGXREPLACE

Parses the regular expression in the first argument against the string in the second argument and replaces the first match with the string in the third argument.

rgxReplace(*regular_expression1* , *string1* , *string2*)

Table 21.64 *Arguments to RGXREPLACE*

Argument	Description
regular_expression1	Specifies a regular expression or a reference to a regular expression in the function context.
string	Specifies a string.
string2	Specifies a string.

```
rgxReplace('(google|yahoo|bing)','http://www.google.com','sas')=http://
www.sas.com
```

RGXREPLACEALL

Parses the regular expression in the first argument against the string in the second argument and replaces any match with the string in the third argument.

rgxReplaceAll(*regular_expression1* , *string1* , *string2*)

Table 21.65 *Arguments to RGXREPLACEALL*

Argument	Description
regular_expression1	Specifies a regular expression or a reference to a regular expression in the function context.

Argument	Description
string	Specifies a string.
string2	Specifies a string.

```
rgxReplaceAll('(google|yahoo|bing)','http://www.google.com/google/
products','sas')
       =http://www.sas.com/sas/products
```

RGXTOKEN

Uses the first argument as a delimiter within the second argument to find all strings separated by that delimiter.

rgxToken(*delimiter* , *string* , <*index* >)

Table 21.66 *Arguments to RGXTOKEN*

Argument	Description
delimiter	Specifies a regular expression or a reference to a regular expression in the function context.
string	Specifies a string.
index	Specifies a numeric value that serves as an index when parsing the string. When the index is less than or equal to the number of tokens in the string, the token at the index value is returned. Otherwise, null is returned. The default value is 0.

```
rgxToken('/','data/opt/sas/dataflux')=data
rgxToken('/','data/opt/sas/dataflux',2)=sas
rgxToken('\.','www.sas.com')=www
```

RGXV

Parses the regular expression in the first argument against the string in the second argument and returns all matches delimited by the specified delimiter.

rgxV(*regular_expression* , *string* , *delimiter* <*group* >)

Table 21.67 Arguments for RGXV

Argument	Description
regular_expression	Specifies a regular expression or a reference to a regular expression in the function context.
string	Specifies a string.
delimiter	Specifies a character value that serves as a delimiter when parsing *string* .
group	Specifies a numeric reference to the regular expression. When specified, the result is the content of the specified numeric regular expression group.

```
rgxV('(jerry|scott|vince)',
            'The ESP product has jerry, scott, and vince working on
it',
            ' : ')=jerry : scott : vince
```

ROUND

Returns the rounded numeric value of the argument.

round(*argument*)

Table 21.68 Arguments for ROUND

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
round(34.56)=35
round(34.46)=34
```

SETCONTAINS

This function has two uses. 1) Parses a specified set of tokens containing the specified delimiter to check whether a specified string appears within the set. 2) References an existing set of tokens in the function context to check whether a specified string appears in the set.

setContains(*set_of_tokens* , *delimiter* , *string*)

setContains(*#reference string*)

Table 21.69 *Arguments for SETCONTAINS*

Argument	Description
set_of_tokens	Specifies a string of tokens.
delimiter	Specifies a character value used as a delimiter.
string	Specifies a string.
reference	Specifies a reference to a function context.

```
setContains('one,two,three,four',',','two')=1
setContains(#mySet,'five')=0
```

STARTSWITH

Returns true when the first argument starts with the second. Otherwise, returns false.

startsWith(*argument1* , *argument2*)

Table 21.70 *Arguments for STARTSWITH*

Argument	Description
argument1 , *argument2*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character.

Argument	Description
	■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
startsWith('www.sas.com','www.')=1
startsWith('www.sas.com','sww.')=0
```

STRING

Returns a concatenated single string value from one or more arguments.

string(*argument* <, *argument2* ,...*argumentN* >)

Table 21.71 *Arguments to STRING*

Argument	Description
argument(s)	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
string(33.9)=33.9
string($id,'-',eventNumber())=eventid-0
```

STRINGLENGTH

Returns the length of the string value of the argument.

stringLength(*argument*)

Table 21.72 Arguments to STRINGLENGTH

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
stringLength('SAS ESP XML')=11
```

STRIP

Returns the string created after removing leading or trailing white space from the argument.

strip(*argument*)

Table 21.73 Arguments to STRIP

Argument	Description
argument	Specifies a string.

```
strip('          SAS ESP XML     ')=SAS ESP XML
```

SUBSTRING

Returns the string created by taking the substring of the value of the first argument at the specified index.

substring(*argument* , *index* , <*length* >)

Table 21.74 Arguments to SUBSTRING

Argument	Description
argument	Specifies a string.

Argument	Description
index	Specifies a numeric value that defines an index with which to parse the *argument*.
length	Specifies a numeric value. When specified, the substring is *length* size, otherwise it contains all characters to the end of the string.

```
substring('www.sas.com',4,3)=sas
substring('www.sas.com',4)=sas.com
```

SUBSTRINGAFTER

Returns the string that results from taking the value of the first argument after an occurrence of the value of the second argument.

substringAfter(*argument1* , *argument2* , <*index* >)

Table 21.75 *Arguments to SUBSTRINGAFTER*

Argument	Description
argument1 , *argument2*	Specifies a string.
index	Specifies a numeric value that defines an index with which to parse *argument1* . When specified, the content after that occurrence is returned.

```
substringAfter('www.sas.com','.')=sas.com
substringAfter('www.sas.com','.',2)=com
```

SUBSTRINGBEFORE

Returns the string that results from taking the value of the first argument before an occurrence of the value of the second argument.

substringBefore(*argument1* , *argument2* , <*index* >)

Table 21.76 *Arguments to SUBSTRINGBEFORE*

Argument	Description
argument1 , *argument2*	Specifies a string.

Argument	Description
index	Specifies a numeric value that defines an index with which to parse *argument1* . When specified, the content after that occurrence is returned.

```
substringBefore('www.sas.com','.')=www
substringBefore('www.sas.com','.',2)=www.sas
```

SUM

Returns the sum of the numeric values of all arguments.

sum(*argument* , *argument2* ...<*argumentN* >)

Table 21.77 *Arguments to SUM*

Argument	Description
argument , *argument2* , ...*argumentN*	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
sum(33,22,55.4,34,min(0,4,-9))=135.4
```

SWITCH

Parses arguments beginning with the second one. When an argument matches the first, it returns the following argument. If no match is found, it returns null.

switch(*argument* , *argument2* , ...<*argumentN* >, <*argumentN+1* >)

Table 21.78 Arguments to SWITCH

Argument	Description
argument , argument2 ...argumentN+1	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
switch('bob','jerry','manager','bob','developer')=developer
switch('jerry','jerry','manager','bob','developer')=manager
switch('moe','jerry','manager','bob','developer')=
```

SYSTEMMICRO

Returns the number of microseconds since Jan 1, 1970.

systemMicro()

```
systemMicro()=1420557039483912
```

SYSTEMMILLI

Returns the number of milliseconds since Jan 1, 1970.

systemMilli()

```
systemMilli()=1420557039483
```

TIMECURRENT

Returns the current time.

timeCurrent()

```
timeCurrent()=1421157236
timeString(timeCurrent())=Tue Jan 13 08:53:56 2015
```

TIMEDAYOFMONTH

Returns the day of the month of the current or specified time.

timeDayOfMonth(<*argument* >)

Table 21.79 *Arguments to TIMEDAYOFMONTH*

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeDayOfMonth()=13
timeDayOfMonth(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=21
```

TIMEDAYOFWEEK

Returns the day of the week of the current or specified time.

timeDayOfWeek(<*argument* >)

Table 21.80 *Arguments to TIMEDAYOFWEEK*

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time. Returns a value of 0 through 6 (Sunday through Saturday).

```
timeDayOfWeek()=2
timeDayOfWeek(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=0
```

TIMEDAYOFYEAR

Returns the day of the year of the current or specified time.

timeDayOfYear(<*argument* >)

Table 21.81 Arguments to TIMEDAYOFYEAR

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeDayOfYear()=12
timeDayOfYear(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=171
```

TIMEGMTTOLOCAL

Converts the GMT that is specified in the argument to local time.

Table 21.82 Arguments to TIMEGMTTOLOCAL

Argument	Description
argument	Specifies a time value or a function that returns a time value.

```
timeString(timeGmtToLocal(timeCurrent()))=Tue Jan 13 02:10:08 2015
```

TIMEGMTSTRING

Writes the GMT time represented by the first argument.

timeGmtString(*argument* , <*argument2* >)

Table 21.83 Arguments to TIMEGMTSTRING

Argument	Description
argument	Specifies a time value or a function that returns a time value.
argument2	Specifies a time format.

```
timeString(timeCurrent(),'%Y-%m-%d %H:%M:%S %Z')=2015-02-20 07:57:18
EST
timeGmtString(timeCurrent(),'%Y-%m-%d %H:%M:%S %Z')=2015-02-20
12:57:18 GMT
```

TIMEHOUR

Returns the hour of the day of the current or specified time.

timeHour(<*argument* >)

Table 21.84 *Arguments to TIMEHOUR*

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeHour()=9
timeHour(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=14
```

TIMEMICRO

Returns the number of microseconds of the supplied argument, or the number of microseconds since Jan 1, 1970 when an argument is not specified.

timeMicro(*argument*)

Table 21.85 *Arguments to TIMEMICRO*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
timeMicro()=1553283757000000
timeMicro(timeParse('06/21/2015 00:00:00','%m/%d/%Y
%H:%M:%S'))=1434859200000000
```

TIMEMILLI

Returns the number of milliseconds of the supplied argument, or the number of microseconds since Jan 1, 1970 when an argument is not specified.

timeMilli(*argument*)

Table 21.86 *Arguments to TIMEMILLI*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you specify this: **`#myXML`**. ■ a function.

```
timeMilli()=1553283876000
timeMilli(timeParse('06/21/2015 00:00:00','%m/%d/%Y
%H:%M:%S'))=1434859200000
```

TIMEMINUTE

Returns the minute of the hour of the current or specified time.

timeMinute(<*argument* >)

Table 21.87 *Arguments to TIMEMINUTE*

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeMinute()=22
timeMinute(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=45
```

TIMEMINUTEOFDAY

Returns the minute of the day of the current or specified time.

timeMinuteOfDay(<*argument* >)

Table 21.88 *Arguments to TIMEMINUTEOFDAY*

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeMinuteOfDay()=563
timeMinuteOfDay(timeParse('06/21/2015 13:45:15','%m/%d/%Y
%H:%M:%S'))=885
```

TIMEPARSE

Returns a string that represents the time specified in the first argument.

timeParse(*time* ,<*format* >)

Table 21.89 *Arguments to TIMEPARSE*

Argument	Description
time	Specifies a time specification or a function that returns a time specification.
format	Specifies a time format, When you do not specify a time format, the default system time format is used. Time formats must conform to those supported by the UNIX strptime() function.

```
timeParse(timeString())=1421159135
timeParse('01/01/2015 00:00:00','%m/%d/%Y %H:%M:%S')=1420088400
```

TIMESECOND

Returns the second of the minute of the current or specified time.

timeSecond(<*argument* >)

Table 21.90 *Arguments to TIMESECOND*

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeSecond()=37
timeSecond(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=15
```

TIMESTAMP

Returns the current time as a string.

timeStamp(<*format* >)

Table 21.91 *Arguments to TIMESTAMP*

Argument	Description
format	Specifies a time format that is supported by the UNIX `strftime` function.

```
timeStamp()=Thu Feb 19 15:09:35 2015
timeStamp('%m-%d-%Y')=02-19-2015
```

TIMESTRING

Returns the time represented by the first argument.

timeString(*time* ,<*format* >)

Table 21.92 *Arguments to TIMESTRING*

Argument	Description
time	Specifies a time specification or a function that returns a time specification.
format	Specifies a time format, When you do not specify a time format, the default system time format is used.

Argument	Description
	Time formats must conform to those supported by the UNIX strftime() function.

```
timeString(timeCurrent())=Thu Feb 19 15:09:35 2015
timeString(timeCurrent(),'%m-%d-%Y')=02-19-2015
```

TIMESECONDOFDAY

Returns the second of the day of the current or specified time.

timeSecondofDay(<*argument* >)

Table 21.93 *Arguments to TIMESECONDOFDAY*

Argument	Description
argument	specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeSecondOfDay()=34035
timeSecondOfDay(timeParse('06/21/2015 13:45:15','%m/%d/%Y
%H:%M:%S'))=53115
```

TIMETODAY

Returns a value that represents the first second of the current day relative to local time.

timeToday()

```
timeToday()=1421125200
```

TIMEYEAR

Returns the number of years since 1900 of the current or specified time.

timeYear(<*argument* >)

Table 21.94 Arguments to TIMEYEAR

Argument	Description
argument	Specifies an expression that defines a specific time, or a function that returns a specific time.

```
timeYear()=115
timeYear(timeParse('06/21/2013 13:45:15','%m/%d/%Y %H:%M:%S'))=113
```

TOLOWER

Converts the value of the argument to lowercase.

toLower(*string*)

Table 21.95 Arguments to TOLOWER

Argument	Description
string	Specifies a string.

```
toLower('Http://Www.Sas.Com/Products/Esp')=http://www.sas.com/
products/esp
```

TOUPPER

Converts the value of the argument to uppercase.

toUpper(*string*)

Table 21.96 Arguments to TOUPPER

Argument	Description
string	Specifies a string.

```
toUpper('Http://Www.Sas.Com/Products/Esp')=HTTP://WWW.SAS.COM/
PRODUCTS/ESP
```

TRANSLATE

For each character in the second argument, finds the corresponding characters in the first argument and replaces them with the corresponding characters in the third.

translate(*argument1* , *argument1* , *argument3*)

Table 21.97 *Arguments to TRANSLATE*

Argument	Description
argument1 , *argument2* , *argument3*	Specifies a string. The length of *argument2* and *argument3* must be identical

```
translate('replace all vowels with its capital
equivalent','aeiou','AEIOU')
          =rEplAcE All vOwEls wIth Its cApItAl EqUIvAlEnt
```

TRUE

Returns true when the Boolean value of the argument is true. Otherwise, it returns false.

true(*argument*)

Table 21.98 *Arguments to TRUE*

Argument	Description
argument	Specifies one of the following: ■ a literal value, either string or numeric. Enclose string values in single or double quotation marks ■ a value to be resolved from an event field. Precede field names with the $ character. ■ a value that refers to a resource. For example, when you use the `xpath` function to refer to an XML object named myXML, you would specify this: **`#myXML`**. ■ a function.

```
true('testing')=1
true('')=0
true(gt(10,5))=1
```

URLDECODE

Decodes the URL represented by the argument.

urlDecode(*argument*)

Table 21.99 *Arguments to URLDECODE*

Argument	Description
argument	Specifies a string.

For more information to encoding and decoding URLs, see this reference.

```
urlDecode('http%3A%2F%2Fwww%2Esas%2Ecom%2Fproducts%2Fevent%20stream
%20processing')
=http://www.sas.com/products/event stream processing
```

URLENCODE

Encodes the URL represented by the argument.

urlEncode(*argument*)

Table 21.100 *Arguments to URLENCODE*

Argument	Description
argument	Specifies a string.

For more information to encoding and decoding URLs, see this reference.

```
urlEncode('http://www.sas.com/products/event stream processing')
=http%3a%2f%2fwww%2esas%2ecom%2fproducts%2fevent%20stream%20processing
```

XPATH

Parses the XML in the first argument, evaluating the second argument in the XML context.

xpath(*argument1* , *argument2* , <*argument3* >)

Table 21.101 Arguments to XPATH

Argument	Description
argument1	Specifies an instance of XML, represented by valid XML textual context or by a reference to XML elsewhere.
argument2	Specifies an evaluation string.
argument3	Specifies a separator used when the function returns multiple results.

```
xpath('<info><name>john smith</name><hobby>running</hobby>
              <hobby>reading</hobby><hobby>golf</hobby></info>',
             './/name/text()')=john smith
xpath(#myXml,'.//hobby/text()',',')=running,reading,golf
```

PART 3

Additional Ways to Interact with Windows

22

Using the Scoring API

Benefits of the Scoring API

The Scoring API enables you to score input events and return scored events on demand.

The Scoring API uses a request-response communication pattern, where a request is sent to a running ESP project and a response is returned. The communication pattern is synchronous: the system waits for the response before continuing. This contrasts with streaming and publish/subscribe communication patterns, where data flows continuously and asynchronously.

You can use the Scoring API to help design and debug ESP projects. The configuration of connectors and adapters depends on the environment in which a project runs. You might use SAS Event Stream Processing Studio to design and test projects in an environment where live data is not available. The Scoring API enables you to simulate data inputs without needing a live data source or in addition to a live data source. You can then use SAS Event Stream Manager to create a project container for your project and deploy it in different environments. For more information, see "Working with Project Containers" in *SAS Event Stream Processing: Using SAS Event Stream Manager*.

Here are some other use cases for the Scoring API:

- Real-time fraud checks. Suppose that a financial institution wants to verify transactions for fraud before processing them. The Scoring API could be used to send a request event that includes fields such as transaction amount, location, and user ID. SAS Event Stream Processing routes the event to a Score window that references a trained fraud detection model. The response event could include a fraud risk score.
- Instant personalization. Suppose that a retail website should offer personalized product recommendations based on user behavior. When a user visits the site and views a product, a scoring request could be sent, with user profile and activity data. SAS Event Stream Processing could apply a personalization model to generate tailored recommendations. The response event could include recommended products.
- On-demand scoring. Suppose that a customer support agent is on a call with a customer and wants to assess the risk of that customer stopping to use the product or service. A scoring request with customer data could be sent, and SAS Event Stream Processing could return a risk score based on a predictive model. The agent could use the score to guide retention strategies.
- Event-driven automation. Suppose that a smart factory wants to trigger maintenance actions when sensor data indicates equipment wear. Sensor data could be sent to SAS Event Stream Processing using the Scoring API. SAS Event Stream Processing could evaluate the data using a predictive maintenance model. If the score exceeds a threshold, further action could be triggered.

Introduction to the Scoring API's Functionality

The scoring request supports multiple input events and multiple output events. The JSON code for the request encloses all inputs and outputs in an array.

Here is an example request:

```
curl -X POST http://myserver:1234/eventStreamProcessing/v2/score --
data-binary @input.json
```

Information relevant to the Scoring API is added to the project's XML code, using the following attributes of the `project` element.

- `score-input-window`
- `score-input-field`
- `score-output-window`
- `score-output-field`

For more information about these attributes, see "project" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

You can use SAS Event Stream Processing Studio to set the project-level attributes. For more information, see "Update Project Properties" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

You can use the test mode in SAS Event Stream Processing Studio to send requests and receive responses. For more information, see "Use the Scoring API When Testing a Model" in *SAS Event Stream Processing: Using SAS Event Stream Processing Studio*.

The score request depends on one or more input events generating one or more output events. If the output events are not seen by the ESP server in a certain amount of time, the request times out and sends back a 500 response code (server error). Here is an example:

```
$ curl -w "response code is %{http_code}\n" "http://myserver:2222/
eventStreamProcessing/v2/score" --data-binary @claim_2.xml
Score request timed out after 20 seconds.
response code is 500
```

The time-out defaults to 20 seconds, but it can be configured in the **esp-properties.yml** file. Here is an example:

```
server:
    score_timeout: 30  # Timeout, in seconds, for Scoring API requests
```

The Request

The request is a POST request whose request body can be in two forms: JSON array or raw data.

The first form is a JSON array where the objects are an array of `name` and `value` children that represent event values:

```
[
    {
        "inputs":
        [
            {"name":"patient","value":"Patient 1"},
            {"name":"statistic","value":"pulse"},
            {"name":"value","value":122},
            {"name":"_opcode_","value":"upsert"}
        ]
    }, {
        "inputs":
        [
            {"name":"patient","value":"Patient 1"},
            {"name":"statistic","value":"systolic_bp"},
            {"name":"value","value":110},
            {"name":"_opcode_","value":"upsert"}
        ]
    }
]
```

Alternatively, the JSON objects can have names and values that represent events:

```
[
    {
        "data": {
            "patient":"Patient 1",
            "statistic":"pulse",
            "value":33,
            "_opcode_":"upsert"
        }
    },{
        "data": {
            "patient":"Patient 1",
            "statistic":"systolic_bp",
            "value":110,
            "_opcode_":"upsert"
        }
    }
]
```

The values in either the `inputs` or `data` objects are the names and values of SAS Event Stream Processing event fields that are published into the Source window that received events for scoring.

You can specify the opcode of the event with the `_opcode_` value. The default opcode is Insert.

The second form of the request body is raw data. This method uses the `score-input-field` attribute of the `project` element to set a single field value in the event data and publish it into the model. This is useful if you are dealing with large, amorphous data (such as images) or large chunks of JSON or XML code. This approach enables you to send the raw data in the request instead of formatting a JSON object with the possible overhead of issues such as Base64 encoding.

This approach can be used only in a Source window whose `autogen-key` attribute is set. In the following example, an image processing model has an entry point that contains an identifier and an image:

```
<window-source index="pi_EMPTY" insert-only="true" autogen-key="true"
name="w_data">
  <schema>
    <fields>
      <field key="true" name="id" type="int64"/>
      <field name="image" type="blob"/>
    </fields>
  </schema>
</window-source>
```

The `project` element looks like this:

```
<project name="annotate" threads="8" score-input-window="w_data" score-
input-field="image" score-output-window="w_annotate">
```

You can send an image into the model directly like this:

```
curl -v -X POST "http://myserver:2222/eventStreamProcessing/v2/score"
--data-binary @cars.jpg -H "content-type: application/octet-stream"
```

The Response

Like the request body, the response body can take two forms: a JSON array containing 0 or more events, or raw data. The response can contain raw data if the following conditions are met:

- The `score-output-field` attribute of the `project` element has been used to specify a field.
- The `score-output-window` attribute has collected a single event.

When these conditions are not met, the response contains a JSON array with elements in one of the two formats mentioned in the request description. Here is an example of a response with an array of `name` and `value` objects:

```
[
    {
        "outputs":[
            { "name":"patient", "value":"Patient 1" },
            { "name":"statistic", "value":"pulse" },
            { "name":"value", "value":122.000000 },
            { "name":"lower", "value":50.000000 },
            { "name":"system", "value":"cardiovascular" },
            { "name":"upper", "value":100.000000 }
        ]
    }, {
        "outputs":[
            { "name":"patient", "value":"Patient 1" },
            { "name":"statistic", "value":"systolic_bp" },
            { "name":"value", "value":110.000000 },
            { "name":"lower", "value":60.000000 },
            { "name":"system", "value":"cardiovascular" },
            { "name":"upper", "value":80.000000 }
        ]
    }
]
```

Here is an example of a response with a single JSON object:

```
[
    {
        "data":{
            "patient":"Patient 1",
            "statistic":"pulse",
            "value":33.000000,
            "lower":50.000000,
            "system":"cardiovascular",
            "upper":100.000000
        }
    }, {
        "data":{
            "patient":"Patient 1",
```

```
            "statistic":"systolic_bp",
            "value":110.000000,
            "lower":60.000000,
            "system":"cardiovascular",
            "upper":80.000000
        }
    }
]
```

The JSON format of the response matches that of the request body. If the request body is in name-value array format, the response is also in that format. If the request body is in object format (or raw format), the response is in JSON object format.

The request always returns a JSON array, even if no events were generated by the input. In this case, an empty JSON array is returned:

```
[]
```

Examples

Claims Processing Example

In this example, a car accident claim is represented as XML code:

```
<Claim>
   <ClaimID>214961</ClaimID>
   <Policy_No>12345</Policy_No>
   <paid_assessors>660.00</paid_assessors>
   <paid_extras>8822.80</paid_extras>
   <paid_investigators>0.00</paid_investigators>
   <claim_value>117777.00</claim_value>
   <Parties>
        <Party>
            <gender>male</gender>
            <NationalID>000001</NationalID>
            <role>PH driver</role>
            <vehicle>2000 Kia Spectra</vehicle>
            <vehicleReg>REG-1</vehicleReg>
            <ClaimID>214961</ClaimID>
            <injuries>
                <injury>CUTS</injury>
                <injury>BRUISES</injury>
            </injuries>
        </Party>
        <Party>
            <NationalID>000002</NationalID>
            <role>TP driver</role>
            <vehicle>1999 Peugeot 504</vehicle>
            <vehicleReg>REG-2</vehicleReg>
```

```
            <ClaimID>214961</ClaimID>
            <injuries>
                <injury>BROKEN LEG</injury>
            </injuries>
        </Party>
        <Party>
            <gender>male</gender>
            <NationalID>000003</NationalID>
            <role>TP passenger</role>
            <vehicle>1999 Peugeot 504</vehicle>
            <vehicleReg>REG-3</vehicleReg>
            <ClaimID>214961</ClaimID>
            <injuries>
                <injury>WHIPLASH</injury>
                <injury>SOFT TISSUE</injury>
            </injuries>
        </Party>
    </Parties>
    <Vehicles>
        <Vehicle>
            <vehicle_make_model>2000 Kia Spectra</vehicle_make_model>
            <vehiclereg>ABC0798</vehiclereg>
            <vehicle_value>88356.05</vehicle_value>
            <vehicle_damage>72467.15</vehicle_damage>
        </Vehicle>
        <Vehicle>
            <vehicle_make_model>1999 Peugeot 504</vehicle_make_model>
            <vehiclereg>XYZ5340</vehiclereg>
            <vehicle_value>43682.25</vehicle_value>
            <vehicle_damage>35826.95</vehicle_damage>
        </Vehicle>
    </Vehicles>
</Claim>
```

The score input window contains a field for an identifier and a string field that holds the raw claim data:

```
<window-source name="claimdata" insert-only="true" autogen-key="true">
  <schema>
    <fields>
      <field name="id" type="int64" key="true"/>
      <field name="claimData" type="string"/>
    </fields>
  </schema>
</window-source>
```

The score output window shows the injuries incurred by the parties in the accident:

```
<window-lua name="injuries" events="create">
  <schema copy="parties">
    <fields>
      <field name="injury" type="string"/>
    </fields>
  </schema>
  <code>...</code>
</window-lua>
```

Because the `claimData` field in the Source window is going to receive the raw XML code, the `project` element looks like this:

```
<project name="claims" threads="4" score-input-window="claimdata"
       score-input-field="claimData" score-output-window="injuries">
```

With this configuration, the following request publishes the claim XML code into the model. The XML code is assumed to be in a file called **`claim.xml`**.

```
$ curl -X POST "http://myserver:2222/eventStreamProcessing/v2/score" --
data-binary @claim.xml
```

The response contains all injuries caused by this claim:

```
[
    {
        "data":{
            "id":"27",
            "claim_id":"214961",
            "vehicle":"2000 Kia Spectra",
            "registration":"REG-1",
            "national_id":"000001",
            "injury":"CUTS"
        }
    },
    {
        "data":{
            "id":"28",
            "claim_id":"214961",
            "vehicle":"2000 Kia Spectra",
            "registration":"REG-1",
            "national_id":"000001",
            "injury":"BRUISES"
        }
    },
    {
        "data":{
            "id":"30",
            "claim_id":"214961",
            "vehicle":"1999 Peugeot 504",
            "registration":"REG-2",
            "national_id":"000002",
            "injury":"BROKEN LEG"
        }
    },
    {
        "data":{
            "id":"32",
            "claim_id":"214961",
            "vehicle":"1999 Peugeot 504",
            "registration":"REG-3",
            "national_id":"000003",
            "injury":"WHIPLASH"
        }
    },
    {
        "data":{
            "id":"33",
```

```
            "claim_id":"214961",
            "vehicle":"1999 Peugeot 504",
            "registration":"REG-3",
            "national_id":"000003",
            "injury":"SOFT TISSUE"
        }
    }
]
```

Image Annotation Example

In this example, raw input and output are used to improve efficiency in image processing. The input Source window has an ID and an image:

```
<window-source name="w_data" index="pi_EMPTY" insert-only="true"
autogen-key="true">
  <schema>
    <fields>
      <field key="true" name="id" type="int64"/>
      <field name="image" type="blob"/>
    </fields>
  </schema>
</window-source
```

The output Score window is a Python window that has an `annotated_image` field that contains a copy of the input image with annotations:

```
<window-python name="w_annotate" events="annotate_event">
  <schema>
    <fields>
      <field key="true" name="id" type="int64"/>
      <field name="annotated_image" type="blob"/>
    </fields>
  </schema>
  <code>...</code>
</window-python>
```

The project element points to the following windows and fields:

```
<project index="pi_EMPTY" name="computer_vision_with_onnx" threads="8"
    score-input-window="w_data" score-input-field="image"
    score-output-window="w_annotate" score-output-
field="annotated_image">
```

For example, you can upload a picture of pelicans flying:

Use the following command to specify to the ESP server that the request body contains binary data and that the client wants to receive binary data:

```
curl -X POST "http://myserver:2222/eventStreamProcessing/v2/score"
    --data-binary @pelicans.jpg -H "content-type: application/octet-
stream" 1
    -H "accept: application/octet-stream" -o annotated.jpg 2
```

1 The `content-type` header tells the server that binary content is being sent from the client.

2 The `accept` header tells the server to return the output image as binary data.

The **curl** command writes the binary response to a local file named `annotated.jpg`, which contains the image with the annotations applied:

Image Processing Example

The Image Processing with Scoring API example in SAS Event Stream Processing Studio GitHub provides a complete project that uses the Scoring API. You can use SAS Event Stream Processing Studio to install the example. For more information, see Install Example Projects.

23

Using the Model Context Protocol

Intended Use and Limitations on Use

Intended Use

SAS Event Stream Processing with Model Context Protocol (MCP) integration is intended to enable customers to expose ESP project functionality as an MCP server, allowing large language models (LLMs) to interact with running ESP projects. This integration provides an interface for LLM-driven applications to query or execute ESP project logic. Customers are responsible for selecting and configuring their own LLMs and managing any associated licensing, quality control, security, and content guardrails. Customers remain responsible for ensuring that

their use of the MCP integration complies with applicable laws, data privacy requirements, and customers' internal governance policies.

The MCP functionality is intended for use by personnel who understand event stream processing concepts and the benefits and risks of MCP integration. MCP does not introduce new capabilities beyond those already available in SAS Event Stream Processing; it changes the access pattern from traditional APIs to MCP-based interaction. Accessing these capabilities via MCP does not automatically add any additional guardrails, input validation, or security controls beyond what the ESP project itself provides. For example, if an ESP project supports operations such as create, read, update, or delete (CRUD), these operations will also be accessible via MCP. Customers should consult the official MCP documentation for detailed guidance on configuration and best practices, particularly regarding security.

Limitations on Use

Generative AI models, including LLMs, use statistical predictions and can generate incorrect or otherwise unexpected or problematic outputs. When customers leverage the MCP integration with the LLM of their choice, those LLMs may produce incomplete or inaccurate responses. Customers should not use the MCP integration without first developing an understanding of the benefits and risks of MCP integration, including as to data integrity, data privacy, and security. Customers should validate all outputs before acting on them and implement appropriate review processes for their specific use cases.

Introduction to Model Context Protocol

The Model Context Protocol (MCP) is a protocol that provides an interface between large language models (LLMs) and external resources (for example, APIs or databases). It provides a standardized way for applications to share contextual information with language models, expose tools to artificial intelligence (AI) systems, and build adaptable workflows.

The Model Context Protocol can be implemented using a local or a remote transport mechanism. The SAS Event Stream Processing server uses the Streamable HTTP transport mechanism to enable remote access to MCP resources. An endpoint such as `/eventStreamProcessing/v2/mcp` enables MCP clients to establish an HTTP connection with the ESP server. After the connection is established, JSON messages are sent and received over the connection. A persistent Server Sent Event (SSE) connection is also established. The SSE connection is a one-way connection from the server to client over which the server can send data. For more information, see Streamable HTTP.

If you have a SAS Event Stream Processing project that includes MCP configuration, the MCP server is exposed when you run the project. Now the project is a resource that an LLM can contact.

There are three major elements in the Model Context Protocol: prompts, resources, and tools. Prompts act as a starting point and define the intent and interaction. Resources provide the context or data. After the model understands the task, it uses tools to perform actions. For more information, see the MCP documentation. With SAS Event Stream Processing, you use MCP tools to configure how a project interacts with LLMs. MCP resources are not supported in this release.

In addition, MCP apps extend core MCP functionality by combining tools and resources to provide interactive user experiences. An app uses tools to perform actions and resources to supply data, context, and user interface content.

The configuration of `mcp` elements in a SAS Event Stream Processing model is specified within a single `mcp` element that goes directly beneath a `project` element. Here is an example of a configured `mcp` element:

```
<project name="myproject" index="pi_EMPTY" threads="4">1
  <mcp>
    <prompts>
        ...
    </prompts>

    <tools>
        ...
    </tools>
    <code snippet="..."><![CDATA[2
        Lua or Python Code
    ]]></code>
  </mcp>
  ...
</project>
```

1 Creates a project called `myproject`.

2 The `code` element is optional. If used, it contains Lua or Python code that is used to support MCP tools. For more information, see "code" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

You can use SAS Event Stream Processing Studio to configure XML elements that are related to MCP. For more information, see Expose a Project as an MCP Server.

Using Prompts

Prompts enable ESP servers to provide messages or instructions on how to interact with language models. They are sent from the ESP server to the client to be displayed to the user. Prompts are controlled by the user and act as guides to the client to point back to the ESP server's capabilities.

Here is an example of a configured prompt:

```
<prompt name="...">
    <title>...</title>
    <description>...</description>
    <text>...</text>
    <fields>
        <field name="...">
            <description>...</description>
        </field>
    </fields>
</prompt>
```

The `prompt` element has a name attribute, title, description, text, and field elements. Here are descriptions of the attributes and elements:

- Name: This attribute is the name of the prompt and must be unique to the model.
- Title: This element specifies a title for the prompt.
- Description: This element specifies a description of what the prompt does.
- Text: This element contains the text that is displayed by the external client through a chat window. Text that is preceded by a `$` character is treated as a variable and prompts the user to enter a value for the variable.
- Fields: This element contains field elements that refer to the variables that are created in the `text` element. Field elements have a name attribute and a `description` element nested within them. If variables were not created in the `text` element, you do not need to define fields.

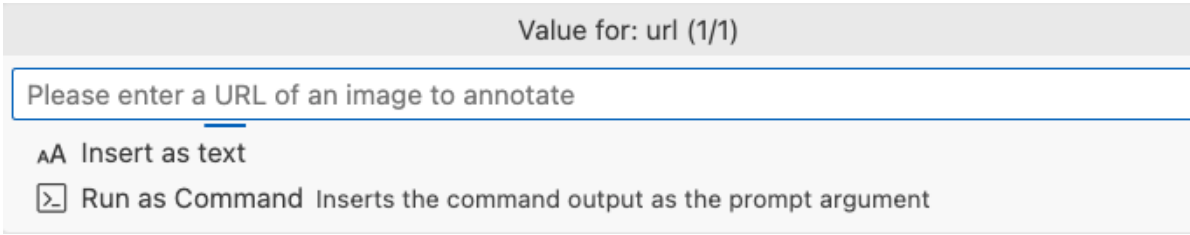
For more information, see "prompts" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Prompt Example

In this example, you have an ESP server with an MCP tool that can perform object detection on an image, annotate the results, and return the annotated image to the client. You can direct the user to call this MCP tool with the following prompt:

```
<prompt name="annotate_image">
    <title>Annotate Image</title>
    <description>Annotate Image</description>
    <text>Please annotate this image $url.</text>
    <fields>
        <field name="url">
            <description>This is the URL of the image to annotate</
description>
        </field>
    </fields>
</prompt>
```

When the user selects the `annotate_image` prompt, the following image appears:



After the URL is entered, the user can attach an image that they want to annotate:

Using Tools

Tools are elements that receive input, perform tasks on the server, and then send results back to the client. Unlike prompts, tools are controlled by the model. This means that the language model can automatically discover and invoke tools based on its contextual understanding and the user's prompts. Each tool must have a type, unique name, title, and description.

The ESP server provides the following tools that can provide services to an MCP client:

- Score tool: This tool enables the client to pass event data into a Source window and get results from the data that passes through the model. The results are returned in the form of event data from a specified output window. A Score tool is configured using `input`, `output`, and `functions` elements. An `input` element contains information about incoming data, and an `output` element contains information about the data that the Score tool generates. A `functions` element contains optional functions that are executed during the course of running the Score tool. For an example of a configured Score tool, see "Score Tool Example".
- Query tool: This tool enables the client to request specific information from the contents of a window. It returns the contents of a specified output window to the LLM directly. A Query tool is configured using an `output` element that contains a `fields` element and a `filter` element. The `field` elements within a `fields` element list the output field names, and the `filter` element is an optional filter that can be applied to the output events. For an example of a configured Query tool, see "Query Tool Example".
- Subscribe tool: This tool enables the client to subscribe to a window in the model and to receive notifications in the form of event data. The Query and Subscribe tools are similar, and they are configured the same way. However, a key difference is that a Subscribe tool does not return events directly to the requesting LLM. For an example of a configured Query tool, see "Subscribe Tool Example".

- Publish tool: This tool enables the client to publish data into an Event Stream Processing model. A Publish tool is configured using an `input` element that contains `field`, `data`, `url`, and `blocksize` elements. The `input` element contains information about incoming data, and its window attribute specifies the Source window to publish events to. For an example of a configured Publish tool, see "Publish Tool Example".
- Custom tool: This tool enables a model designer to write custom Lua or Python code that accepts input and generates results that are returned to the client. A Custom tool is configured using `output` and `functions` elements. The `output` element contains information about data that the Custom tool generates, and the `functions` element specifies the functions to call. The ESP server calls the function and creates the appropriate response to return to the client. For an example of a configured Custom tool, see "Custom Tool Example".

For more information about elements and attributes for different tool types, see "XML Language Elements Relevant to Model Context Protocol" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Examples of MCP Tools

Score Tool Example

Here is an example of a configured Score tool:

```
<tool type="score" name="..."> 1
    <title>...</title>
    <description>...</description>
    <config>
      <input window="..." array="..."> 2
        <fields>
          <field name="..." mime-type="...">
            <description>...</description>
          </field>
          ...
        </fields>
      </input>
      <output window="..." array="..."> 3
        <fields>
          <field name="..." mime-type="...">
            <description>...</description>
          </field>
          ...
        </fields>
      </output>
      <functions> 4
        <function name="pre" code="..." />
        <function name="post" code="..." />
        <function name="events" code="..." />
          <use>...</use>
        </function>
```

```
      </functions>
    </config>
  </tool>
```

1 The type is set to `score` and the tool is given a unique name.

2 Information about the input data is specified. This includes the name of the Source window that receives the data and whether the tool is receiving an array of objects.

3 Information about the output data is specified. This includes the name of the downstream window to monitor for events and the name of the output array if the tool returns an array of objects.

4 Any functions that are executed during the course of running the Score tool are called here. The value of `name` determines when the function is run.

For more information about elements and attributes for the Score tool, see "tools" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Query Tool Example

Here is an example of a configured Query tool:

```
<tool type="query" name="">1
    <title>...</title>
    <description>...</description>
    <config>
      <output window="...">2
        <fields>
          <field name="..." type="...">
            <description>...</description>
          </field>
          ...
        </fields>
        <filter function="...">3
          <fields>
            <field name="..." type="..." optional="..." array="...">
              <description>...</description>
            </field>
            ...
            </field>
          </fields>
        </filter>
      </output>
    </config>
  </tool>
```

1 The tool type is set to `query` and the tool is given a unique name.

2 Information about the data that the Query tool generates is specified. This includes the window from which the tool reads events.

3 An optional filter is defined in order to filter the output events. The Lua or Python function to filter the events must be specified.

For more information about elements and attributes for the Query tool, see "tools" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Subscribe Tool Example

Here is an example of a configured Subscribe tool. It is identical to a configured Query tool with the exception of the type attribute:

```
<tool type="query" name="">1
    <title>...</title>
    <description>...</description>
    <config>
      <output window="...">2
        <fields>
          <field name="..." type="...">
            <description>...</description>
          </field>
          ...
        </fields>
        <filter function="...">3
          <fields>
            <field name="..." type="..." optional="..." array="...">
              <description>...</description>
            </field>
            ...
            </field>
          </fields>
        </filter>
      </output>
    </config>
  </tool>
```

1 The tool type is set to `subscribe` and the tool is given a unique name.

2 Information about the data that the Subscribe tool generates is specified. This includes the window from which the tool reads events.

3 An optional filter is defined in order to filter the output events. The Lua or Python function to filter the events must be specified.

For more information about elements and attributes for the Subscribe tool, see "tools" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Publish Tool Example

Here is an example of a configured Publish tool:

```
<tool type="publish" name="...">1
    <title>...</title>
    <description>...</description>
    <config>
```

```
      <input window="..."> 2
        <field>...</field>
        <data>...</data> 3
        <url>...</url> 4
        <blocksize>...</blocksize> 5
      </input>
    </config>
  </tool>
```

1 The tool type is set to `publish` and the tool is given a unique name.

2 Information about the data that the Publish tool generates is specified. This includes the name of the Source window that receives the input data.

3 The `data` element is optional and contains the contents of the data to be published. If you define a `data` element, you do not need to define a `url` element.

4 The `url` element is optional and contains a URL with the contents of the data to be published. If you define a `url` element, you do not need to define a data element.

5 The `blocksize` element is optional and specifies the number of events that you want to put into each event block when the data is published.

For more information about elements and attributes for the Publish tool, see "tools" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Custom Tool Example

Here is an example of a configured Custom tool:

```
<tool type="custom" name="getProcInfo"> 1
      <title>Get Process Information</title>
      <description>Get process information including CPU and memory
usage information.</description>
      <config>
        <output>
          <fields>
            <field name="path" type="string" /> 2
            <field name="cmdline" type="string" />
            <field name="created" type="string" />
            <field name="user" type="string" />
            <field name="cwd" type="string" />
            <field name="status" type="string" />
            <field name="cpu" type="number" />
            <field name="num_threads" type="number" />
            <field name="memory" type="object"> 3
              <fields>
                <field name="resident" type="number"/>
                <field name="virtual" type="number"/>
                <field name="shared" type="number"/>
                <field name="text" type="number"/>
                <field name="data" type="number"/>
              </fields>
```

```
            </field>
          </fields>
        </output>
        <functions>
          <function name="call" code="getProcInfo"/> 4
        </functions>
      </config>
    </tool>
```

1 The tool type is set to `custom` and the tool is given a unique name.

2 Each `field` element that is defined correlates to a field in the output JSON file. Fields must have unique names.

3 If a field's `type` attribute is set to `object`, you can nest fields inside your `field` element.

4 A function named `getProcInfo` is run when it is called during the course of running the Custom tool.

The `getProcInfo` function is defined in a separate code element:

```
<code><![CDATA[

import datetime
import psutil 1

def getProcInfo(parms):
    data = {}
    p = psutil.Process()
    data["path"] = p.exe()
    data["cmdline"] = p.cmdline()
    data["created"] =
datetime.datetime.fromtimestamp(p.create_time()).strftime("%Y-%m-%d
%H:%M:%S")
    data["user"] = p.username()
    data["cwd"] = p.cwd()
    data["status"] = p.status()
    data["cpu"] = p.cpu_percent(interval=.5)
    data["num_threads"] = p.num_threads()
    tmp = p.memory_info()
    data["memory"] =
{"resident":tmp[0],"virtual":tmp[1],"shared":tmp[2],"text":tmp[3],"data
":tmp[5]}
    print(str(data))
    return(data)

  ]]></code>
```

1 The Custom tool uses the Python `psutil` package to get information and return it to the client.

For more information about elements and attributes for the Custom tool, see "tools" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Addition Example

This example shows a complete project with a Score tool that takes an array of integers and adds them together. The results are sent back to the client.

```
<project name="p" pubsub="auto" threads="4">
  <mcp>
    <prompts>
      <prompt name="add_3_numbers">1
        <title>Add 3 Numbers</title>
        <description>Use SAS Event Stream Processing to add an
arbitrary number of integers and return the result.</description>
        <text>Can you add these numbers $num1, $num2, and $num3 and
return the sum?</text>2
        <fields>3
          <field name="num1">
            <description>The first number</description>
          </field>
          <field name="num2">
            <description>The second number</description>
          </field>
          <field name="num3">
            <description>The third number</description>
          </field>
        </fields>
      </prompt>
    </prompts>
    <tools>
      <tool type="score" name="add_numbers">4
        <title>Add Numbers</title>
        <description>Use SAS Event Stream Processing to add an
arbitrary number of integers and return the result.</description>
        <config>
          <score-functions>
            <events function="handleEvent" />
          </score-functions>
          <input window="s">5
            <fields>
              <field name="numbers" type="number" array="true">
                <description>The numbers to add</description>
              </field>
            </fields>
          </input>
          <output window="lua">6
            <fields>
              <field name="sum">
                <description>the sum</description>
              </field>
            </fields>
          </output>
```

```
        </config>
      </tool>
    </tools>
    <code><![CDATA[ 7
def handleEvent(data):
  response = "The sum of "
  i = 0
  for value in data["numbers"]:
    if i == len(data["numbers"]) - 1:
      response += " and "
    elif i > 0:
      response += ", "
    response += str(value)
    i += 1
  response += " is " + str(data["sum"])
  return response
    ]]></code>
  </mcp>
  <contqueries>
    <contquery name="cq"> 8
      <windows>
        <window-source name="s" autogen-key="true" insert-
only="true"> 9
          <schema>
            <fields>
              <field name="id" type="int64" key="true"/>
              <field name="numbers" type="array(i32)"/>
            </fields>
          </schema>
        </window-source>
        <window-lua name="lua" events="create"> 10
          <schema>
            <fields>
              <field name="id" type="int64" key="true"/>
              <field name="numbers" type="array(i32)"/>
              <field name="sum" type="int32"/>
            </fields>
          </schema>
          <code><![CDATA[
            local   id = 1

            function create(data)
                local event = {}
                local sum = 0

                for _,value in ipairs(data.numbers)
                do
                  sum = sum + value
                end

                event.id = id
                event.numbers = data.numbers
                event.sum = sum

                id = id + 1
```

```
                return event
              end
            ]]></code>
          </window-lua>
        </windows>
        <edges>
          <edge source="s" target="lua"/> 11
        </edges>
      </contquery>
    </contqueries>
</project>
```

1 Creates a prompt called `add_3_numbers`.

2 Specifies the text to show the user and creates the variables `num1`, `num2`, and `num3`.

3 Creates the fields that prompt the user to enter values for the variables.

4 Creates a score tool called `add_numbers`.

5 Specifies the configuration of the input data.

6 Specifies the configuration of the output data.

7 Contains the code that defines the `handleEvent` function.

8 Creates a continuous query called `cq`.

9 Defines the score input Source window.

10 Defines the output window.

11 Creates an edge with a leading edge of `s` and a trailing edge of `lua`.

Using Apps

Overview

Apps enable MCP servers to provide interactive user interfaces to MCP clients such as GitHub Copilot and Claude Desktop. For more information about the specifications for MCP apps, see apps.mdx.

These apps enable users to view and interact with data that is processed within an ESP model. By connecting data sources to visual representations, MCP apps allow real-time information to be presented in a dynamic way.

An app definition consists of a name, description, and a configuration that defines the event sources (ESP windows) from which to receive data. The definition also includes visual components to display the data. Every app has a configuration element that defines its characteristics. The content of the element can be embedded in the XML file, or it can exist in an external URL. If the configuration is

stored in an external URL, you can modify the app configuration and run the updated app without restarting the ESP server. For more information about the element, see "XML Language Elements Relevant to Model Context Protocol" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Event Sources

Event sources define where an app retrieves its data within an ESP model. They are configured using elements, where each event source specifies a name and the ESP window from which to obtain data. Each event source is either a collection or a stream. A *collection* represents the full set of events currently in the window, and a *stream* delivers events as they are created.

Event sources include an interval that controls how frequently data is delivered, and they can include a filter to limit the events returned based on specified criteria. Here is an example of an app that uses four different event sources:

```
<event-sources>
    <event-source name="shipdata" type="collection"
window="Boat_Current" interval="250"/>
    <event-source name="speeding" type="stream"
window="Last_Speeding_Event" interval="250"/>
    <event-source name="exclusion" type="stream"
window="Last_Exclusion_Violation" interval="250"/>
    <event-source name="sourcewin" type="stream" window="Boat"
interval="250"/>
</event-sources>
```

Visual Components

Visual components enable you to create visual representations of event data within an app. They are used to display data from defined sources in formats such as charts, tables, or maps.

Each visual component is associated with a data source and uses configuration settings to control how the data is displayed and how the visualization behaves. Apps can include one or more visual components, allowing data to be presented across multiple views within the same app. Visual components support interactive features:

- "Table"
- "Bar Chart"
- "Line Chart"
- "Time Series Chart"
- "Compass"
- "Gauge"

- "Pie Chart"
- "Image Viewer"
- "Geo Map"
- "Model Viewer"
- "Log Viewer"
- "Metrics Viewer"
- "Custom Component"

Examples of Visual Components

Table

The table component enables you to display data in a tabular form. It supports the following property values:

- title: The title of the component.
- values: The values to be displayed in the table. The default behavior is to display all values.
- image_width: The width (in pixels or a percentage) of any images in the table.
- image_height: The height (in pixels or a percentage) of any images in the table.

Here is an example of a configured table:

```
<component source="shipdata" type="table">1
    <properties>
      <property name="title">Current Ship Data</property>2
      <property
name="values">boat_id,heading,speed,latitude,longitude</property>3
    </properties>
</component>
```

1 Creates a table component that uses data from `shipdata`.

2 Defines the title of the component as `Current Ship Data`.

3 Defines the values to be displayed in the table. The specified values appear as the column headings.

Bar Chart

The bar chart component enables you to display data as a vertical or horizontal bar chart. It supports the following property values:

- title: The title of the component.
- y: Numeric event values to plot on the chart. The values determine the length of the bars.

- orientation: The orientation of the chart. Valid values are **`horizontal`** and **`vertical`**.
- calculate_range: A Boolean that indicates whether the chart automatically sets the axis minimum and maximum values using the data values. The default value is **`false`**.
- range: The range of possible values written as a comma-separated list of two values.

Here is an example of a configured bar chart:

```
<component source="shipdata" type="bar">1
    <properties>
      <property name="title">Current Ship Speed</property>
      <property name="y">speed</property>2
      <property name="orientation">horizontal</property>
      <property name="range">0,8</property>3
    </properties>
</component>
```

1 Creates a bar chart component that uses data from `shipdata`.

2 Determines the bar chart to display each ship's speed.

3 Defines the range of values from zero to eight.

Line Chart

The line chart component enables you to display one or more numeric values as points connected by lines to show variations in data. It supports the following property values:

- title: The title of the component.
- y: Numeric event values to plot on the chart. The values are plotted as points connected by lines.
- orientation: The orientation of the chart. Valid values are **`horizontal`** and **`vertical`**.
- markersize: The size of the markers in the chart. The default value is **`0`**.
- line_width: The width of the line that connects the points. The default value is **`4`**.
- curved: A Boolean that indicates whether to draw curved lines. The default value is **`false`**.
- calculate_range: A Boolean that indicates whether the chart automatically sets the axis minimum and maximum values using the data values. The default value is **`false`**.
- range: The range of possible values written as a comma-separated list of two values.

Here is an example of a configured line chart:

```
<component source="shipdata" type="line">1
    <properties>
      <property name="title">Current Ship Speed</property>
      <property name="y">speed</property>
```

```
      <property name="markersize">20</property> 2
      <property name="line_width">10</property> 3
      <property name="range">0,8</property>
    </properties>
</component>
```

1 Creates a line chart component that uses data from `shipdata`.

2 Sets the marker size to 20.

3 Sets the line width to 10.

Time Series Chart

The time series chart component enables you to display data over time. You must have a date or time field in the data to configure this component. It supports the following property values:

- title: The title of the component.
- time: The date or time field to use for the X axis of the chart.
- (Optional) series: A field used to group the data into separate lines. If not specified, all data points are displayed as a single line.
- y: The name of the field that contains the value to plot.
- markersize: The size of the markers in the chart. The default value is **`0`**.
- line_width: The width of the line that connects the points. The default value is **`4`**.
- curved: A Boolean that indicates whether to draw curved lines. The default value is **`false`**.
- calculate_range: A Boolean that indicates whether the chart automatically sets the axis minimum and maximum values using the data values. The default value is **`false`**.
- range: The range of possible values written as a comma-separated list of two values.
- max: The maximum number of data points to be displayed for each series. The default value is **`50`**.

Here is an example of a configured time series chart:

```
<component source="shipdatastream" type="timeseries"> 1
    <properties>
      <property name="title">Ship Speed Over Time</property>
      <property name="series">boat_id</property> 2
      <property name="y">speed</property>
      <property name="time">@timestamp</property> 3
      <property name="range">0,8</property>
    </properties>
</component>
```

1 Creates a time series chart component that uses data from `shipdatastream`.

2 Specifies to group the data by `boat_id` and to create separate lines for each ID.

3 Specifies that the `timestamp` field is the X axis of the chart.

Compass

The compass component enables you to display data with directional information. The compass displays each event by key and uses a specified value field to show direction. It supports the following property values:

- title: The title of the component.
- value: The field that contains the compass heading values and the direction of the needle.
- (Optional) size : The value that specifies the diameter of each compass. The default value is `250`.

Here is an example of a configured compass:

```
<component source="shipdata" type="compass">1
    <properties>
      <property name="title">Current Ship Heading</property>
      <property name="value">heading</property>2
    </properties>
</component>
```

1 Creates a compass component using data from `shipdata`.

2 Specifies `heading` as the compass heading values.

Gauge

The gauge component enables you to display a single numeric event field. Create multiple gauges to compare numeric events fields side by side. The component supports the following property values:

- title: The title of the component.
- value: The name of the field that contains the value to be displayed in the gauge.
- range: The range of the gauge written as a comma-separated list of two values. The default range is `0,10`.
- (Optional) size: The size of each gauge. The default value is `250`.

Here is an example of a configured gauge:

```
<component source="shipdata" type="gauge">1
    <properties>
      <property name="title">Current Ship Speed</property>
      <property name="value">speed</property>2
    </properties>
</component>
```

1 Creates a gauge component that uses data from `shipdata`.

2 Specifies the `speed` field as the value to be displayed.

Pie Chart

The pie chart component enables you to display numeric values as slices that represent portions of a whole. It supports the following property values:

- title: The title of the component.
- value: The value to be displayed in the pie chart.

Here is an example of a configured pie chart:

```
<component source="violations" type="pie">1
    <properties>
      <property name="title">Speeding Violations</property>
      <property name="value">count</property>2
    </properties>
</component>
```

1 Creates a pie chart component that uses data from `violations`.

2 Specifies `count` as the value to be displayed in the pie chart.

Image Viewer

The image viewer component enables you to display event data that contains images. It supports the following property values:

- title: The title of the component.
- image: The name of the image field to be displayed.
- scale: A floating-point value that is used to scale the image.

Here is an example of a configured image viewer:

```
<component type="imageviewer" source="annotate">1
    <properties>
      <property name="title">Images</property>
      <property name="image">image</property>2
      <property name="scale">.35</property>3
    </properties>
</component>
```

1 Creates an image viewer component that uses data from `imageviewer`.

2 Specifies `image` as the field to be displayed.

3 Specifies to scale the image to 35% of its original size.

Geo Map

The geo map component enables you to display event data that contains geographic information, and it uses LeafletJS to display the information. It supports the following property values:

- title: The title of the component.
- lat: The name of the event field that is used for the latitude value.

- lon: The name of the event field that is used for the longitude value.
- (Optional) center: The geographic center of the map in the format `lat:<latitude>,lon:<longitude>`.
- (Optional) zoom: The zoom level of the map. A low zoom level shows entire continents, and a high zoom level shows the details of a city.
- (Optional) clicktips: A Boolean value that indicates whether to use persistent tooltips. Tooltips appear when a data item is selected.
- (Optional) delegate: The JavaScript object that contains functions used to control the geo map display. The delegate object contains the `init`, `properties`, `html`, and `tooltip` functions.
  - `init(component)`: Initializes the geo map.
  - `properties(map,item)`: Sets the display properties.
  - `html(map,item)`: Returns HTML to display as the map marker.
  - `tooltip(map,item)`: Returns the tooltip of an item.

Here is an example of a configured geo map:

```
<component source="shipdata" type="map">1
    <properties>
      <property name="title">Ship Map</property>
      <property name="delegate">delegate</property>2
      <property name="lat">latitude</property>
      <property name="lon">longitude</property>
      <property name="clicktips">true</property>3
      <property name="center">lat:56.000241,lon:-3.424250</property>
    </properties>
    <css><![CDATA[
    ...
    ]]></css>
    <javascript><![CDATA[

    var delegate = {
    ...
    }
    ]]></javascript>
</component>
```

1 Creates a geo map component that uses data from `shipdata`.

2 Declares the `delegate` Javascript object.

3 Specifies that the geo map uses persistent tooltips.

Model Viewer

The model viewer component enables you to view the windows and edges that make up the model in the server. Use the toolbar to zoom in, zoom out, or reset the model to its original perspective. Hovering over a window enables you to see information about it (for example, type, index, and class). When you select a window, you can view the current event contents, event stream, or XML definition of the window. The component supports the following property value:

- title: The title of the component.

Here is an example of a configured model viewer:

```
<component type="modelviewer">
  <properties>
    <property name="title">Ships Model Viewer</property>
  </properties>
</component>
```

Log Viewer

The log viewer component enables you to display server log information and provides controls for managing logging contexts and clearing the log. Use the toolbar to show server logging or to clear the log. It supports the following property value:

- title: The title of the component.

Here is an example of a configured log viewer:

```
<component type="logviewer">
  <properties>
    <property name="title">Ships Log Viewer</property>
  </properties>
</component>
```

Metrics Viewer

The metrics viewer component enables you to view the ESP metrics that exist in a server. Each metric is displayed as a graphic. Use the toolbar to view the metric window that contains the type, name, and current state of all of the metrics that exist in the server. It supports the following property value:

- title: The title of the component.

Here is an example of a configured metrics viewer:

```
<config>
        <component type="metricviewer">
          <properties>
            <property name="title">Metric Viewer</property>
          </properties>
        </component>
      </config>
```

Custom Component

The custom component enables you to create custom MCP components. These components are written using ESP utilities or other technologies (for example, HTML or JavaScript). It supports the following property values:

- html: Contains the HTML to be included with the component.
- html-url: Contains the external HTML to be included with the component.
- delegate: The JavaScript object that contains functions used to control the custom component. The delegate object contains the `init` function that is

called when the component is initialized. This function does all of the required setup for the component.

Note: If you require external resources for the custom component, you must add the `<resource-domains>` and `<connect-domains>` elements. For more information, see "resource-domains" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*.

Here is an example of a configured custom component:

```
<component type="custom">1
      <properties>
        <property name="title">My Custom Component</property>
        <property name="html"> <![CDATA[  2
        <div style='width:100%;height:100%;display:flex;justify-
content:center;align-items:center;font-size:2rem'>
        This is my simple custom component.
        </div>
        ]]></property>
      </properties>
    </component>
```

1 Creates a custom component.

2 Specifies the HTML code to be included in the component.

App Types and Testing

Visual App

A Visual app displays a single visual component and is configured with any event sources that you want to include. Here is an example of a configured Visual app:

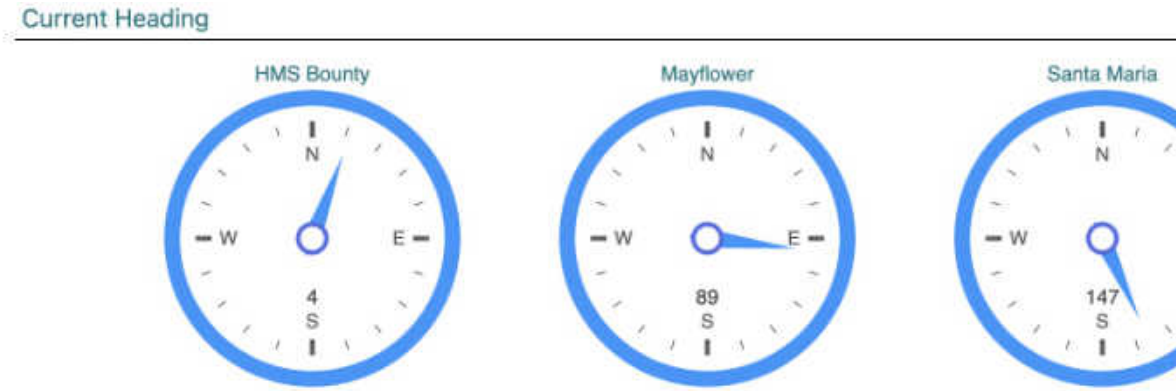
```
<app name="Ships_Heading" type="visual">1
  <title>Compasses Showing Ship Heading</title>
  <description>Compasses Showing Ship Heading</description>
  <config>
    <event-sources>
      <event-source name="shipdata" type="collection"
window="Boat_Current" interval="250"/>2
    </event-sources>
    <component source="shipdata" type="compass">3
      <properties>
        <property name="title">Current Heading</property>
        <property name="value">heading</property>
      </properties>
    </component>
  </config>
</app>
```

1 Creates a Visual app called `Ships_Heading`.

2 Specifies the `event-source` element called `shipdata`, which delivers data every 250 milliseconds.

3 Creates a compass component based on the data from `shipdata`.

The code renders the following Visual app:



Dashboard App

The Dashboard app displays multiple components and is configured with any event sources that you want to include. Dashboard apps are useful when you want to display related components side by side. You can specify the dimensions for each component to determine the layout of the page. Here is an example of a configured Dashboard app with three tables:

```
<app name="Ships_DataDashboard" type="dashboard">1
  <title>Dashboard App for Ship Data.</title>
  <description>Dashboard App for Ship Data.</description>
  <config>
    <event-sources>2
      <event-source name="shipdata" type="collection"
window="Boat_Current" interval="250"/>
      <event-source name="speeding" type="stream"
window="Last_Speeding_Event" interval="250"/>
      <event-source name="exclusion" type="stream"
window="Last_Exclusion_Violation" interval="250"/>
    </event-sources>
    <components>3
      <component name="boat_table" type="table" source="shipdata"
width="40%" height="40%">
        <properties>
          <property
name="values">boat_id,heading,speed,latitude,longitude</property>
          <property name="title">Ship Info</property>
        </properties>
      </component>
      <component name="speeding" type="table" source="speeding"
width="40%" height="50%">
```

```
        <properties>
          <property
name="values">dateTime,boat_id,Max_speed,Location_Name</property>
          <property name="title">Last Speeding Event</property>
        </properties>
      </component>
      <component name="exclusion" type="table" source="exclusion"
width="40%" height="250px">
        <properties>
          <property
name="values">boat_id,Location_Name,Minimum_Distance</property>
          <property name="title">Last Exclusion Violation</property>
        </properties>
      </component>
    </components>
  </config>
</app>
```

1 Creates a Dashboard app called `Ships_DataDashboard`.

2 Specifies three `event-source` elements called `shipdata`, `speeding`, and `exclusion`. All of the event sources deliver data every 250 milliseconds.

3 Creates three table components called `boat_table`, `speeding`, and `exclusion`.

The code renders the following Dashboard app:

Ship Info

boat_id	heading	speed	latitude	longitude
HMS Bounty	270.81	3.908	56.006	-3.45
Mayflower	85.95	3.438	56.015	-3.355
Santa Maria	270.29	4.489	55.999	-3.396

Last Speeding Event

dateTime	boat_id	Max_speed
2017-09-02T19:15:06.000Z	HMS Bounty	3.889
2017-09-02T19:15:08.000Z	HMS Bounty	3.889

Last Exclusion Violation

boat_id	Location_Name	Minimum_Distance
HMS Beagle	Rosyth Dockyard exclusion zone	657
HMS Beagle	Rosyth Dockyard exclusion zone	657

Navigation App

The Navigation app displays multiple components and is configured with any event sources that you want to include. Navigation apps are useful when you want to display multiple components in groupings or tabs. Tabs are displayed at the bottom of the app with a name and optional icon. For more information, see "tabs" in *SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models*. Here is an example of a configured Navigation app with three tabs:

```
<app name="Ships_NavApp" type="navigation">1
  <title>Navigation App for Ship Data.</title>
  <description>Navigation App for Ship Data.</description>
  <config>
```

```
    <event-sources> 2
      <event-source name="shipdata" type="collection"
window="Boat_Current" interval="250"/>
      <event-source name="sourcewin" type="stream" window="Boat"
interval="250"/>
    </event-sources>
    <tabs> 3
      <tab name="data" text="Ship Data" icon="Boat"/>
      <tab name="server" text="Server" icon="Server"/>
      <tab name="source" text="Source Data" icon="Boat"/>
    </tabs>
    <components> 4
      <component name="info" text="Ship Info" type="table"
source="shipdata" width="40%" height="250px" icon="GridView"
tab="data">
        <properties>
          <property
name="values">boat_id,heading,speed,latitude,longitude</property>
          <property name="title">Ship Info</property>
        </properties>
      </component>
      <component name="headings" text="Heading" source="shipdata"
type="compass" icon="ArrowSmallUp" tab="data">
        <properties>
          <property name="title">Headings</property>
          <property name="value">heading</property>
        </properties>
      </component>
      <component source="shipdata" text="Speed" type="gauge"
icon="GrooveDashboard" tab="data">
        <properties>
          <property name="title">Current Speed</property>
          <property name="value">speed</property>
        </properties>
      </component>
      <component name="speed" text="Speed Over Time" type="timeseries"
source="shipdata" icon="TimeSeriesChart" tab="data">
        <properties>
          <property name="title">Ship Speed</property>
          <property name="series">boat_id</property>
          <property name="y">speed</property>
          <property name="time">@timestamp</property>
          <property name="range">0,10</property>
          <property name="max">25</property>
          <property name="markersize">2</property>
        </properties>
      </component>
      <component name="model" type="modelviewer" text="Model Viewer"
icon="OrgChart" tab="server">
        <properties>
          <property name="zoom">true</property>
        </properties>
      </component>
      <component name="log" type="logviewer" text="Log Viewer"
icon="Log" tab="server">
        <properties>
```

```
          <property name="title">Log Viewer</property>
        </properties>
      </component>
      <component name="src" type="table" source="sourcewin"
tab="source">
        <properties>
          <property name="values">boat_id,heading,speed</property>
          <property name="title">Boat Stream Data</property>
        </properties>
      </component>
    </components>
  </config>
</app>
```

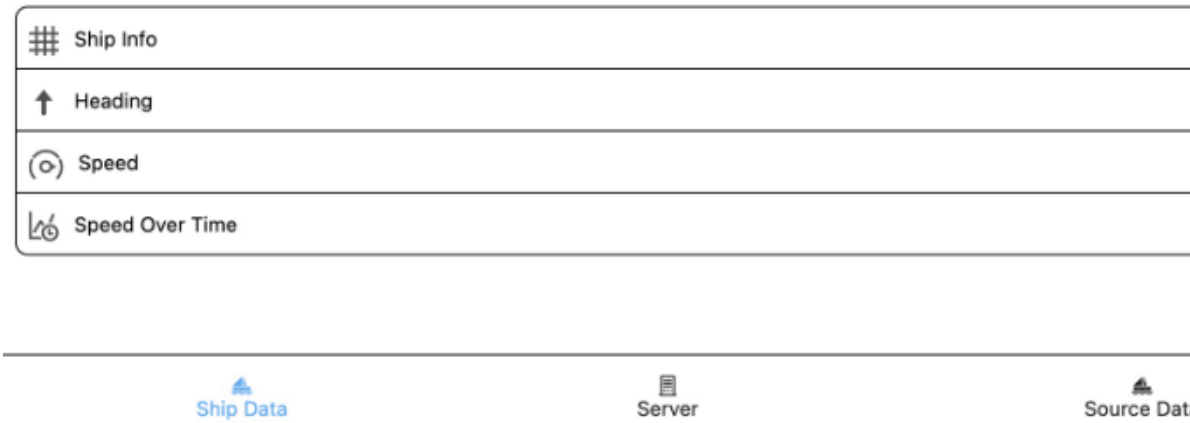
1 Creates a Navigation app called `Ships_NavApp`.

2 Specifies two `event-source` elements called `shipdata` and `sourcewin`. Both of the event sources deliver data every 250 milliseconds.

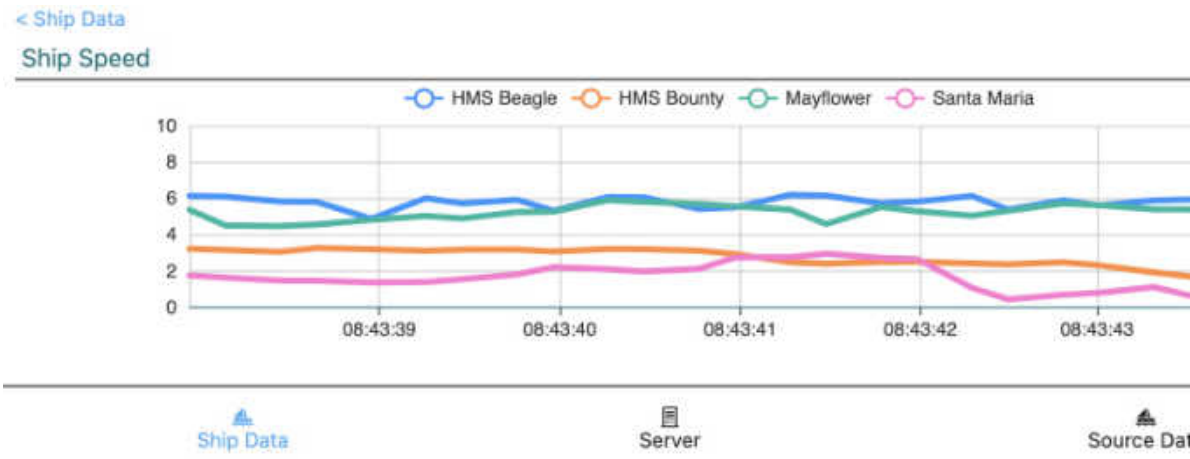
3 Specifies three tabs called `data`, `server`, and `source` to group the components.

4 Creates components that are displayed in the Navigation app. Each component must have an associated tab. Use the `tab` attribute to set the value to the name of one of the defined tabs.

The code renders the following Navigation app:



Because there are four components associated with the **Ship Data** tab, they are displayed as a list. You can select each component to expand it. Use the navigation link to return to the list of components:



If you navigate to a tab that has a singular component associated with it, the component is displayed immediately: