SAS® Event Stream Processing 6.1: Creating and Using Windows

Window Types

An event stream processing model specifies how input event streams are transformed and analyzed into meaningful resultant event streams. Every model contains an engine. An engine contains one or more projects, and each project contains one or more continuous queries.

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

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

Table 1 *Window Types*

Window Type	Description
Source window	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.
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.
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.

Window Type	Description
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.
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.
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).
Remove State window	Facilitates the transition of a stateful part of a model to a stateless part of a model.
Text Category window	Enables you to categorize a text field in incoming events. A Text Category window is Insert-only. The text field could generate zero or more categories with scores. This object enables users who have licensed SAS Contextual Analysis to use its MCO files to initialize a Text Category window.
Text Context window	Enables the abstraction of classified terms from an unstructured string field. This object enables users who have licensed SAS Contextual Analysis to use its Liti files to initialize a Text Context window. Use this window type to analyze a string field from an event's input to find classified terms. Events generated from those terms can be analyzed by other window types. For example, a Pattern window could follow a text context window to look for tweet patterns of interest.
Text Sentiment window	Determines the sentiment of text in the specified incoming text field and the probability of its occurrence. The sentiment value is "positive," "neutral," or "negative." The probability is a value between 0 and 1. A Text Sentiment window is Insert-only. This object enables users who have licensed SAS Sentiment Analysis to use its SAM files to initialize a Text Sentiment window.
Text Topic window	Run SAS Text Miner analytics on events. Text topic windows receive and process text from documents as string fields. Text mining analytics models enter a text topic window through an analytic store file.
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.

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

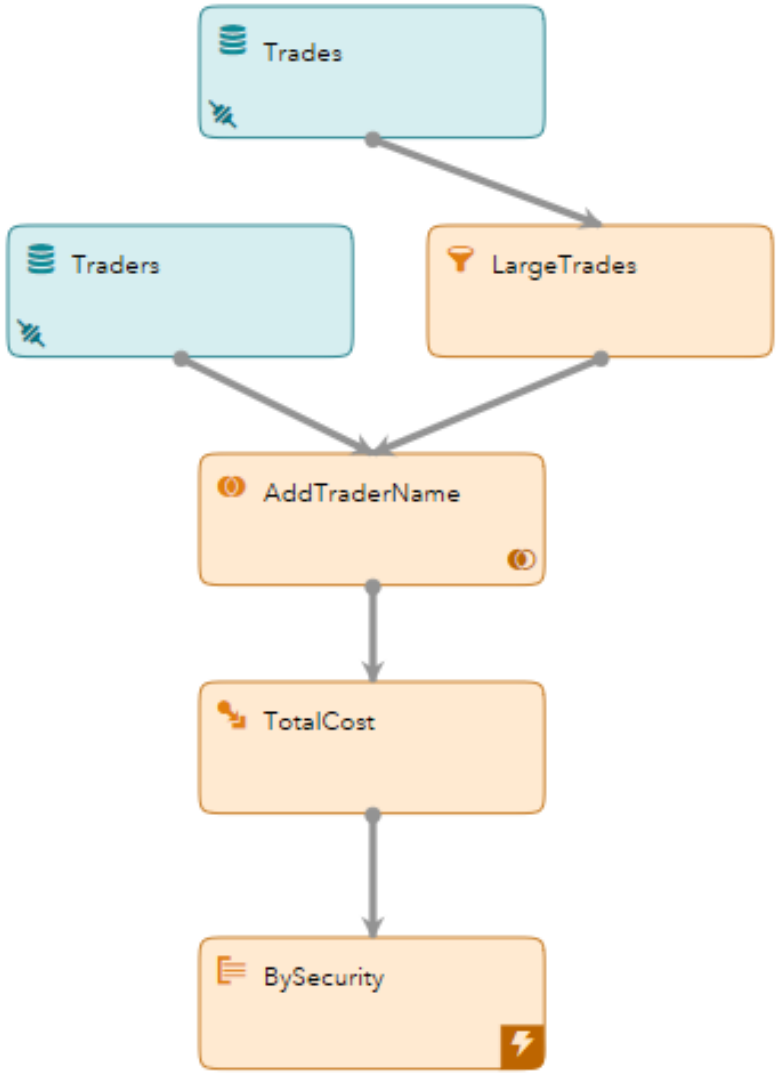
Creating a Continuous Query with Windows and Edges

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 model in SAS Event Stream Processing Studio.

Figure 1 *Continuous Query Diagram*



Here is the corresponding SAS Event Stream Processing XML modeling language. You can find this code in **$DFESP_HOME/examples/xml/trades_xml**.

```
<engine name='trades' port='54321' <!--1-->
    <projects>
        <project name='trades_project' pubsub='auto' threads='4'> <!--2-->
            <contqueries>
                <contquery name='trades_cq'> <!--3-->
                    <windows>
```

1 An engine named trades on a publish/subscribe port of 54321 is established.

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

3 A continuous query named trades_cq is established.

```
<window-source name='Trades' 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>
</projects>
</engine>
```

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

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

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 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

Using Expressions

Overview to Expressions

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

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

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

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

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

For information about how to specify expressions, refer to the *Expression Language: Reference Guide*.

Note: SAS Event Stream Processing uses a subset of the functionality that is documented for the Expression Engine Language. This subset is robust for the needs of event stream processing.

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 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>
```

You can find examples in **`$DFESP_HOME/examples/xml/splitter_initexp_xml`**, **`$DFESP_HOME/examples/xml/splitter_udf_xml`**, and **`$DFESP_HOME/examples/xml/regex_xml`**.

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 2 *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 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 (second granularity)	Constant milliseconds in dfExpressions not supported
Int32 (32 bit)	Integer (64 bit)	64-bit conversion for dfExpressions
Int64 (64 bit)	Integer (64 bit)	64-bit, no conversion
double (64 bit IEEE)	real (192 bit fixed decimal)	real 192-bit fixed point, double 64-bit float
money (192 bit fixed decimal)	real (192 bit fixed decimal)	192-bit fixed point, no conversion

Using Event Metadata in Expressions

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

Table 3 *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 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 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. For more information about global functions, see *Expression Language: Reference Guide*.

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

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

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

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

Using SAS Data Quality Functions

Event stream processing expressions support the use of the SAS Data Quality functions. The following functions are fully documented in the *Expression Language: Reference Guide*:

- dq.case
- dq.gender
- dq.getlasterror

- dq.identify
- dq.initialize
- dq.loadqkb
- dq.matchcode
- dq.pattern
- dq.standardize

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.

Using Source Windows

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.

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

Defining Event Index Types

Source windows can have a primary index and a retention index. There are seven 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 by the engine 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 value is an incremental count (0, 1, 2, ...). For STRING keys, the 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

An engine can contain a metering window to track 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`** in **`esp-properties.yml`**. 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 engine 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 Dictionary*.

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 Dictionary* .

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 group-by fields of **G**.
3. Each non-key field of the output schema is computed by calling an aggregate function with the group **G** as input. The aggregate function computes a scalar value for the corresponding non-key field.
4. The correct output event is generated and output.

Using Aggregate Functions

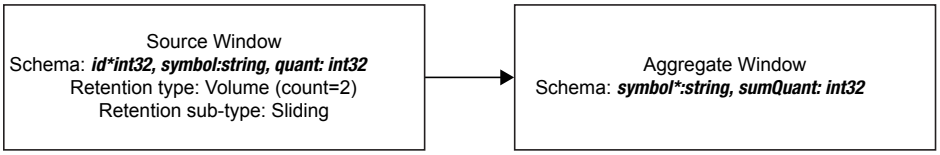
Overview to Using Aggregate Functions

During aggregation, events that enter an Aggregate window are placed into a group based on the Aggregate window's key fields. The aggregate functions provided with SAS Event Stream Processing are run on each group 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 values and collapse the group to 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 *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 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. This means that if these are the only functions used in an Aggregate window, special optimizations are performed. Speed-ups of an order of magnitude in the Aggregate window processing can occur. Some functions are additive regardless of the opcode of the incoming event. Others are additive only when they get Inserts.

Table 4 *Aggregate Functions*

Aggregate Function	Additive	Additive for Inserts	Returns
`ESP_aAve(fieldname)`	True	True	Average of the group
`ESP_aCat(fieldname, stringConstant)`	False	True	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.
`ESP_aCount()`	True	True	Number of events in the group
`ESP_aCountDistinct(fieldname)`	True	True	Number of distinct non-null values in the column specified by field name within a group
`ESP_aCountNonNull(fieldname)`	False	True	Number of events with non-null values in the column for the specified field name within the group
`ESP_aCountNull(fieldname)`	False	True	Number of events with null values in the column for the specified field name within the group
`ESP_aCountOpcodes(opcode)`	True	True	Count of the number of events matching opcode for group
`ESP_aFirst(fieldname)`	False	True	First event added to the group
`ESP_aGUID()`	True	True	A unique identifier
`ESP_aLag(fieldname,lag_value)`	False	True	A lag value where the following holds: ■ `ESP_aLag(fieldname,0) == ESPaLast(fieldname)` ■ `ESP_aLag(fieldname,1)` returns the second lag. This is the previous value of *fieldname* that affected the group.
`ESP_aLast(fieldname)`	False	True	Field from the last record that affected the group

Aggregate Function	Additive	Additive for Inserts	Returns
`ESP_aLastNonNull(fieldname)`	False	True	Field from the last record with non-null value that affected the group
`ESP_aLastOpcode()`	True	True	The opcode of the last record to affect the group
`ESP_aMax(fieldname)`	False	True	Maximum of the group
`ESP_aMin(fieldname)`	False	True	Minimum of the group
`ESP_aMode(fieldname)`	True	True	Mode, or most popular of the group
`ESP_aStd(fieldname)`	True	True	Standard deviation of the group
`ESP_aSum(fieldname)`	True	True	Sum of the group
`ESP_aWAve(weight_fieldname, payload_fieldname)`	True	True	Weighted group average

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 Aggregation window at any time. For this case, the Aggregation 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 Aggregation 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 also 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 group-by 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:

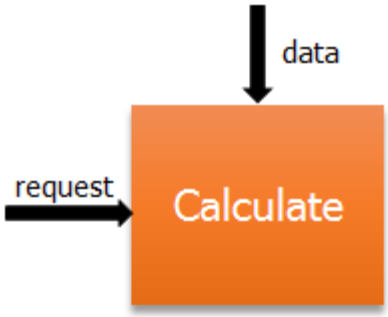
```
U    N    SAIA    15.2872    1100    15.549867    1100
```

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 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 publishes newly transformed score data into output events. (No role is assigned to the outgoing edges, so they 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.

For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

Migrating from a Procedural Window to a Calculate Window

Support for SAS Micro Analytic Service (MAS) modules and stores has moved from the Procedural window to the Calculate window.

To migrate from a Procedural Window to a Calculate Window requires minimal change to your XML code.

Example Code 1 *Procedural Window*

```
<window-procedural name='pw_01'>
                …
</window-procedural>
<edge source='w_source' target='pw_01'/>
```

Example Code 2 *Calculate Window*

```
<window-calculate name='pw_01' algorithm='MAS'>
              …
</window-calculate>
<edge source='w_source' target='pw_01' role='data'/>
```

You can use the **`dfesp_xml_migrate`** command to convert Procedural windows to Calculate windows in your model code. However, you must manually convert DS2 code in table server mode to code that uses SAS Micro Analytic Service modules before running the tool. For more information about this command, see "Migrating XML Code across Product Releases" in *SAS Event Stream Processing: XML Language Dictionary*.

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 Dictionary* and "XML Language Elements Relevant to Compute and Filter Windows" in *SAS Event Stream Processing: XML Language Dictionary* .

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**.

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 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 Dictionary* .

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 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 Dictionary* .

You cannot configure the schema for a Counter window; it is hardcoded as follows:

```
"input*:string,totalCount:int64,totalSeconds:double,totalRate:double,intervalCount:
int64,intervalSeconds:double,intervalRate:double" .
```

The value of *input* is the name of the window that sent the event to the Counter window.

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.

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

```
<window-counter name='counter'
                       count-interval='2 seconds'
                       clear-interval='30 seconds'/>
   <event opcode='upsert' window='trades/trades/counter'>
      <value name='input'>trades</value>
      <value name='intervalCount'>288215</value>
      <value name='intervalRate'>144108</value>
      <value name='intervalSeconds'>2</value>
      <value name='totalCount'>794312</value>
      <value name='totalRate'>132385</value>
      <value name='totalSeconds'>6</value>
   </event>
```

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

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

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. Streamviewer can subscribe to the Counter window to show the results. Alternatively, you can add the Counter window to the trace attribute of the `<contquery>` element that prints formatted events to standard output.

Using Filter Windows

Filter windows use expressions, user-defined functions (global functions), and registered plug-in functions to determine what input events are permitted to stream through. These functions and expressions are called *filter conditions*.

For more information about available filter conditions, see “Overview to Expressions”.

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

Using Functional Windows

Overview to Functional Windows

Use a Functional window to call different types of functions in order to manipulate or transform event data. You define a schema and then a *function context* that contains functions and supporting entities such as regular expressions, XML, and JSON. When an event enters a Functional window, the window looks for a function with

a name that corresponds to each field in its schema. If the function exists, then it is run. The resulting value is entered into the output event. If no function is specified for a field, and a field with the same name exists in the input schema, then the input value is copied directly into the output event.

For information about the XML elements associated with Functional windows, see "window-functional" in *SAS Event Stream Processing: XML Language Dictionary* and "XML Language Elements Relevant to Functional Windows" in *SAS Event Stream Processing: XML Language Dictionary* .

For an example showing how to use Functional windows to monitor stock trades, go to the SAS Event Stream Processing Learning Center.

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

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

For a complete reference on support functions available for defining functions in functional windows, see "Functional Window and Notification Window Support Functions".

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 name space 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 name space.

Understanding and Using Function Context

Overview to Function Context

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

Types of Functions You Can Use

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

- general functions
- functions specific to event stream processing

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

Table 5 *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.

Using Expressions

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

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

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

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

Table 6 *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 Type	Description
`<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>
```

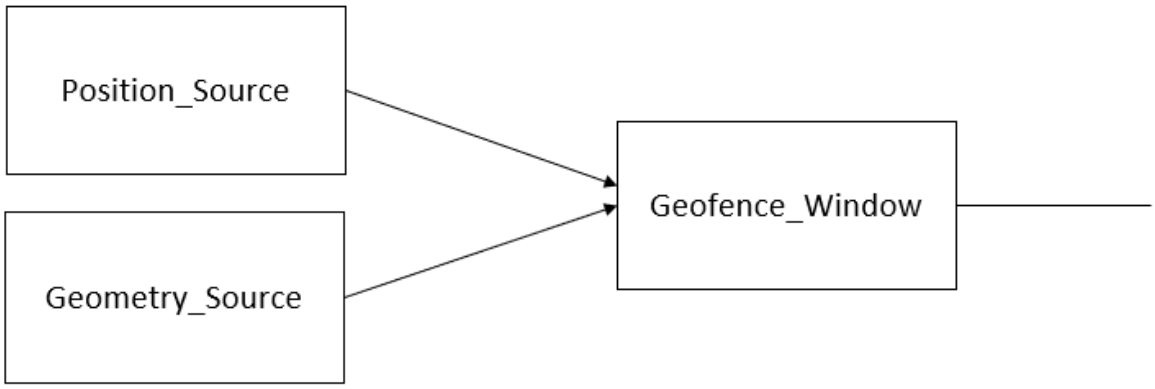
Using Geofence Windows

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 Dictionary* and "XML Language Elements Relevant to Geofence Windows" in *SAS Event Stream Processing: XML Language Dictionary* .

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:

When the polygon is not closed and does not contains any closed ring, it is considered a polyline.

- A single key field of type int64 or string. This field defines the ID of the geometry.
- A data field of type string, rstring, or array(dbl). This field contains the list of the ring's position coordinates. The coordinates are defined as numbers separated by spaces in the X, Y order, separated by spaces when using a field of type string and rstring.
- 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`**.

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

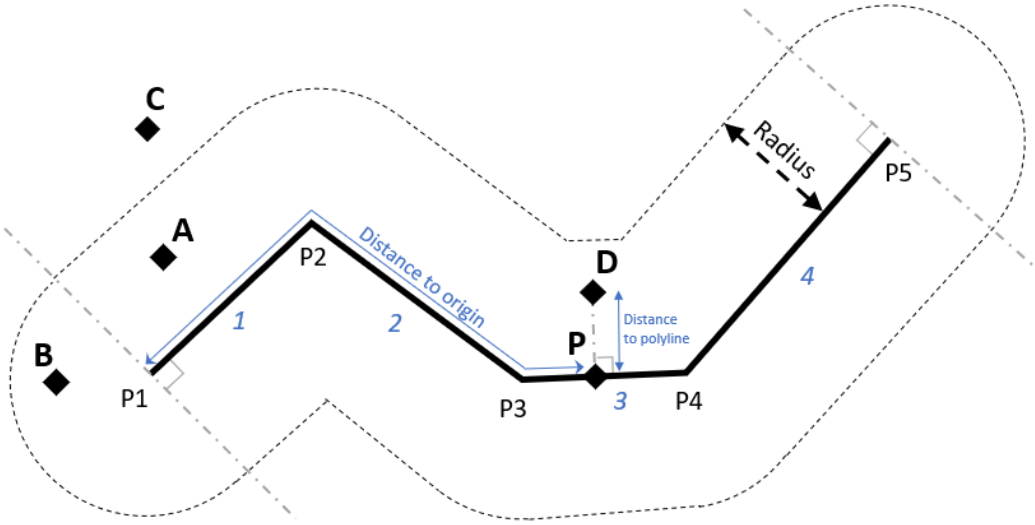
A polyline is a shape that represents a border or a trip wire. Define a polyline as a list of coordinates that represent the polyline's segment or segments. The sequence of the points in the segment definition defines the polyline's vector direction and its left and right side.

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 trip wire 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 *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 Dictionary* .

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 geometry.
- 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.

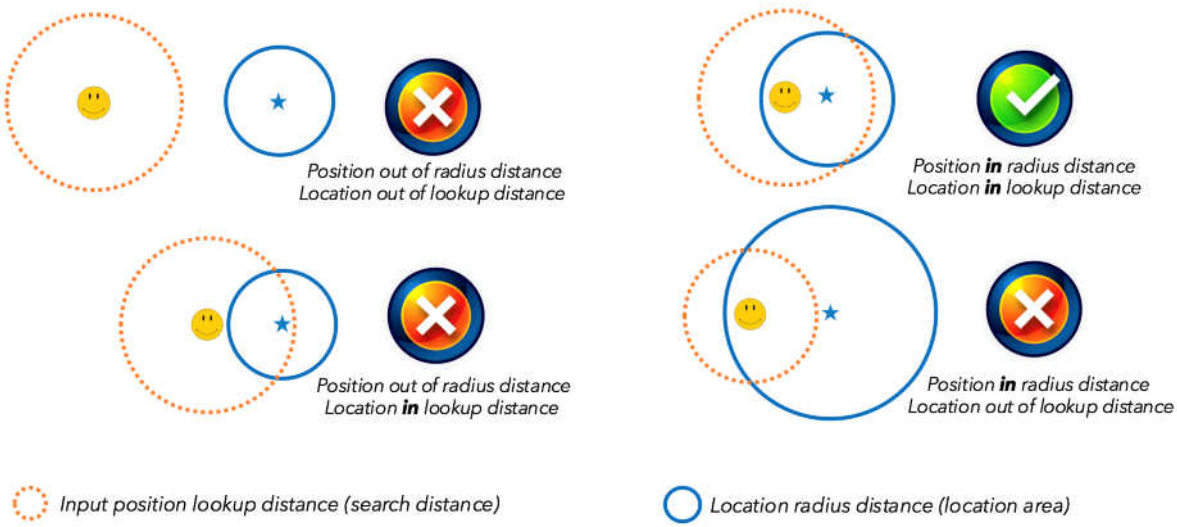
All other fields are ignored.

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 5 *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 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>
```

Because an engine is based on primary keys and supports Inserts, Updates, and Deletes, there are some restrictions placed on the types of joins that can be used. The 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 Dictionary* and "XML Language Elements Relevant to Join Windows" in *SAS Event Stream Processing: XML Language Dictionary* .

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
			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 Indices

For allowed one-to-many and many-to-one joins, a change to the fact table enables immediate lookup of the matching record in the dimension table through its primary index. All key values of the dimension table are mapped in the join 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 indices 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 indices.

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

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>
```

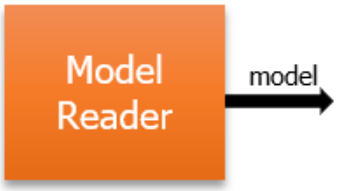
This shows one-to-one mapping of input fields to join non-key fields. You can use calculation expressions and functions to generate the non-key join fields using arbitrarily complex combinations of the input fields.

For XML and C++ code examples of full projects, go to **`$DFESP_HOME/examples`** on Linux systems. Go to **`%DFESP_HOME%\examples`** on Windows systems.

Using Model Reader Windows

In most cases, *Model Reader windows* receive request events that include the location and type of an offline model. Offline models are specified, developed, trained, and stored separately from the ESP server. Model Reader windows publish a model event that contains the model to Score windows or to Model Supervisor windows.

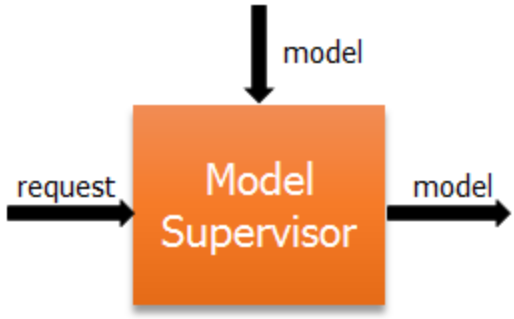
For recommender systems, you can specify offline model property values within the Model Reader window itself. The window publishes a model event based on those values.



For more information, see *SAS Event Stream Processing: Using Streaming 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.

The `<window-model-supervisor>` element has three properties:

- `name=` specifies the name of the window
- `deployment-policy=`

  `immediate`
  sends model events to any receiving window immediately after receiving them

  `on-demand`
  sends model events according to the requests specified at the command line
- `capacity=` specifies the number of current model events to keep. After the capacity is met, older model events are discarded.

For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

Using Notification Windows

Overview to Notification Windows

Use a Notification windows to send notifications through email using the Simple Mail Transfer Protocol (SMTP), text using the Short Message Service (SMS), or multimedia messages using the Multimedia Messaging Service (MMS). These windows, like Functional windows, enable you to define a function context to transform incoming events before processing them for possible notifications.

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

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

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

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

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

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

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.ec.rr.com'
            user='esptest@ec.rr.com'
            password='esptest1' port='587' />
```

Notification Window Delivery Channels

Overview to Notification Window Delivery Channels

The Notification window uses three types of delivery channel:

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

Using the Email Delivery Channel

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

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

The `email` element contains the following attributes:

- `throttle-interval` specifies a time period in which at most one notification is sent to a recipient

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

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

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

- the `sender` email address
- the `recipients` to whom the email message is sent. 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 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>[code]</sender>
      <subject>[code]</subject>
      <from>[code]</from>
      <gateway>[code]</gateway>
      <phone>[code]</phone>
   </sms-info>
   <sms-contents>
      <text-content name=''>...</text-content>
   </sms-contents>
</sms>
```

The `sms` element contains the following attributes:

- `throttle-interval` specifies a time period in which at most one notification is sent a recipient.
- `test` is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

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

The `sms-info` element contains functions or hardcoded values that represent the data to be used to send an SMS 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>[code]</sender>
      <subject>[code]</subject>
      <gateway>[code]</gateway>
      <phone>[code]</phone>
   </mms-info>
   <mms-contents>
      <text-content name=''>...</text-content>
      <image-content name=''>...</image-content>
      ...
   </mms-contents>
</mms>
```

The `mms` element contains the following attributes:

- `throttle-interval` specifies a time period in which at most one notification is sent to a recipient.
- `test` is a Boolean attribute that specifies whether to run in test mode. When running in test mode, the notification is not sent but written to the console. This can be useful when drafting notification messages.

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

The `mms-info` element contains functions or hardcoded values that represent the data to be used to send an 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 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 Dictionary*.

Using Object Tracking Windows

Overview to Object Tracking Windows

Use an Object Tracking window to perform multi-object tracking (MOT) in real time. MOT is the process of accurately estimating the identity and position of multiple objects over time using a model that is based on incoming observations. These observations are the output of an object detection model.

The challenges of multi-object tracking include scene clutter, target dynamics, intra-class and inter-class variation, measurement noise, and frame rate. The Object Tracking window addresses these challenges by coupling trackers with detectors in a paradigm called tracking-by-detection. Specifically, it uses an intersection-over-union (IOU) method of tracking-by-detection that is explained in Bochinski, Eiselein, and Sikora (2017).

For information about the XML elements associated with Object Tracking windows, see window-object-tracker and "XML Language Elements Relevant to Object-Tracking Windows" in *SAS Event Stream Processing: XML Language Dictionary* .

The Intersection Over Union (IOU) Method of Tracking-By-Detection

The IOU method tracks objects using only detection data. Using this method enables incremental and event-oriented tracking. It has produced good results on the MOTChallenge and the UA-DETRAC benchmarks.

The IOU method implemented by the Object Tracking window provides the following functionality:

- It uses a velocity vector to predict the next position of the tracked object when it is missing in the next frame. The `velocity-vector-frames` parameter defines the number of frames used 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 new track only when their IOU with existing tracks is less than the value of `iou-sigma-dup`. This 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, it dies. The parameter `max-track-lives` sets the life duration of tracks without detection.

Input Schema for Object Tracking Windows

The Object Tracking window uses incoming detection data in order to perform multi-object tracking. It uses an input schema that consists of the following fields to propose detections:

- The detected objects count for the event.
- For each detected object, the following fields:
  - The label of the detected object.
  - The detection score.
  - The x coordinate of the bounding box center or the lower corner, respective to the origin of the coordinates. In object detection output, this 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 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 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 often refers to the bottom right corner.
- All other fields are propagated to the output event, including the event key fields.

For example:

```
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
```

The input element of the Object Tracking window defines the object's input fields names convention. 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, ….`

If the `<input>` element is not used, then the standard analytic store output field name conventions are used.

Output Schema for Object Tracking Windows

The output of an Object Tracking window conforms to a wide or a long schema, depending of the value of the `mode` attribute of the output element.

For wide mode, the output schema includes all non-object fields of the input window. This includes the input key fields. Then, for each tracked object, the output schema includes the following fields:

```
prefix_density        :int32
prefixN_id            :int64 1 2 3
prefixN_label         :string
prefixN_score         :double
prefixN_x             :double
prefixN_y             :double
prefixN_w             :double
prefixN_h             :double
prefixN_center_x      :double 4
prefixN_center_y      :double
prefixN_track_x       :array(dbl) 5
prefixN_track_y       :array(dbl)
```

1 Use the `prefix` attribute of the `<output>` element to define the value of `prefix`. The default value of `prefix` is **`'Object'`**.

2 The `N` value is the 0-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 last detection.

5 The `<..>_track_x` and `<..>_track_y` fields contain the list of coordinates of the center of the bounding box of the successive object's positions.

When `velocity-vector="true"`, the following additional fields appear in the output schema:

```
prefixN_vvect_x     :double 1
prefixN_vvect_y     :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 specified in "Input Schema for Object Tracking Windows" , then the output schema becomes the following:

```
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
```

For long mode, the output schema includes all non-object fields of the input window. This includes the input key fields. After those fields, the output schema includes the following:

```
prefix_density              :int32
prefix_id              :int64        key='true'
prefix_label           :string
prefix_score           :double
prefix_x               :double
prefix_y               :double
prefix_w               :double
prefix_h               :double
prefix_center_x        :double
prefix_center_y        :double
prefix_track_x         :array(dbl)
prefix_track_y         :array(dbl)
```

When `velocity-vector="true"`, the output schema contains the following additional fields:

```
prefix_vvect_x        :double
prefix_vvect_y        :double
```

The value of `prefix` is defined by the `prefix` attribute of the output element. The default value of `prefix` is **`'Object'`**.

For example, when `prefix='Object'`, `velocity-vector="true"`, and the input schema is the one specified in "Input Schema for Object Tracking Windows" , the output schema becomes the following:

```
frame_id*:int64,image:blob,objCount:int32,time:stamp,Object_density:int32,Object_id*:int64,Object_label:string,
Object_score:double,Object_x:double,Object_y:double,Object_w:double,Object_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
```

Reference

Bockinski, Erik, Eiselein, Volker, and Sikora, Thomas. "High-Speed Tracking-by-Detection Without Using Image Information." IEEE International Conference on Advanced Video and Signal-based Surveillance, August 2017, Lecce, Italy.

Using Pattern Windows

Overview of Pattern Windows

Use a Pattern window to detect events of interest (EOIs) as they stream through. To create a Pattern window, specify the list EOIs and assemble them into an expression that uses logical operators. The expression can also include temporal conditions.

Specify EOIs by providing the following:

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

Table 7 *Valid Logical Operators for Pattern Logic*

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

To apply a temporal condition to the `fby` function, append the condition to the function inside braces. For example, specify

```
fby{1 hour}(event1,event2)
```

when event2 happens within an hour of event1. Specify

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

when event3 happens within ten minutes of event2, and event2 happens within ten minutes of event1.

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

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

Creating Patterns

Here is an XML example of a pattern from a broker surveillance model. EOIs are specified within `event` elements. The pattern logic is specified within a `logic` element.

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

```
        </output>
    </pattern>
    <pattern>
        <events>
            <event name='e1'>((buysellflg==0) and (broker == seller)
            and (s == symbol) and (b == broker))</event>
            <event name='e2'>((buysellflg==0) and (broker != seller)
            and (s == symbol) and (b == broker))</event>
        </events>
        <logic>fby{10 minutes}(e1,e2)</logic>
        ...
    </pattern>
</patterns>
</window-pattern>
```

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

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

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

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

```
<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.

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 operator and for the first matching EOI causes the variable bindings. Take special care in this case, as this is rarely what you intend when you write a pattern.

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

```
<pattern>
   <events>
      <!-- Someone buys or sells IBM at price > 150.00 and also
           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>fby{5 seconds}(e1, e2)</logic>
...
</pattern>
```

Now 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 both sides of the "and" by the single input 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 also
           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>fby{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. This requires two unique events for the pattern match to occur.

Suppose that you specify a temporal condition for an AND operator such that event `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

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 8 *Expressions and Associated Outcomes*

Expression	Outcome
`not EOI`	A Boolean negation. This 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 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
`notoccur EOI`	Becomes TRUE on application of an event that does not satisfy the *EOI*, but it is not marked FIXED. This implies that more events can be applied to it. As soon as it sees an event that matches the *EOI*, it becomes FALSE and FIXED
`notoccur OPT`	Not permitted
`EO or EO`	An event that is always applied to all non-FIXED sub-trees. It becomes TRUE when one of its two sub-trees become TRUE. It becomes FALSE when both of the sub-trees becomes FASLE. It is FIXED when one of its sub-trees is TRUE and FIXED 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. It becomes TRUE when both of its two sub-trees become TRUE. It becomes FALSE when one of the sub-trees becomes FALSE. It is FIXED when one of its sub-trees is FALSE and FIXED or both of its sub-trees are TRUE and FIXED

Expression	Outcome
`EO FBY EO`	Attempts to complete the left hand side (LHS) with the minimal number of event applications before applying events to the right hand side (RHS). The apply rule is as follows: ■ If the LHS is not TRUE or FALSE, apply event to the LHS until it become TRUE or FALSE. ■ When the LHS becomes FALSE, set the followed by state to FALSE and become FIXED. ■ 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`	bBecomes TRUE on the application of an event that satisfies the EOI and FALSE otherwise. Becomes FIXED on the first application of an event.
`is OPT`	Not permitted

Table 9 *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 might be an input window generating updates or deletions, then you must place a Procedural window between the input window and the Pattern window. The Procedural window then filters out or transforms non-insert data to insert data.

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

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

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

  For example, the following `addEvent` statement would be flagged as invalid:

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

  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 the following:

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

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

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

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

Using Stateless Pattern Windows

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

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

Pattern windows are automatically marked as insert-only. They reject records that are not 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 it needs to be explicitly told that it is insert-only, using the `dfESPwindow::setInsertOnly()` call. This causes the Source window to reject any events with an opcode other than Insert, and permits an index type of `pi_EMPTY` to be used.

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

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

Enabling Pattern Compression

When an event affects a pattern and partially completes it, the event is stored in the pattern instance for future use. When a pattern event completes through a later sequence of events, the stored event is accessed. When the system has an exceptionally large number of partially completed patterns, a large amount of memory might be required the associated stored events. To address this issue, you can compress partially completed patterns and then uncompress them upon pattern completion.

There are two ways to enable pattern compression on projects:

- In C++, call `dfESPproject::setPatternCompression(true)` before a project is started.
- In XML, use the `compress-open-patterns='true'` attribute on a `project` element.

Pattern compression can be useful when a project has a very large number of open patterns waiting for possible completion. It can decrease pattern memory usage by as much as 40% at the expense of a slight increase in CPU usage.

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

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 can improve performance because the Pattern window has fewer events to evaluate.

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 in one of two ways:

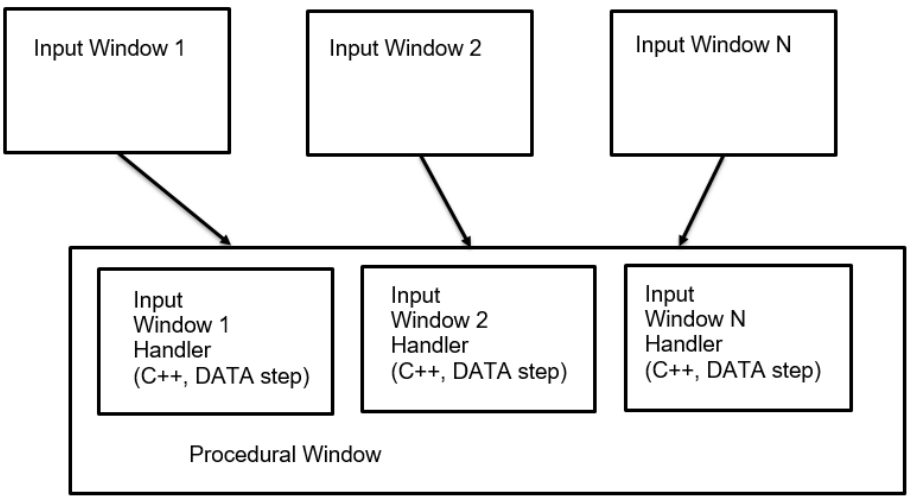
- Use a callable C++ function in a shared library. Use the `cxx-plugin` XML element to define this type of input handler.
- Use DATA step code that calls into an external SAS session that runs on the same server that produces one output row for each input row.

  Note: You must configure Base SAS 9.4 to use UTF-8 before running DATA step code in a Procedural window.

Note: Support for SAS Micro Analytic Service (MAS) modules and stores has moved from the Procedural window to 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.

Figure 6 *Procedural Window with Input Handlers*



In order for the state of the Procedural window to be shared across handlers, an instance-specific context object (such as `dfESPpcontext`) is passed to the handler function. Each handler has full access to what is in the context object. The handler can store data in this context for use by other handlers, or by itself during future invocations.

For information about the XML elements associated with Procedural windows, see "window-procedural" in *SAS Event Stream Processing: XML Language Dictionary* and "XML Language Elements Relevant to Procedural Windows" in *SAS Event Stream Processing: XML Language Dictionary* .

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).

Using DATA Step Window Handlers

Overview

When you write a handler using DATA step statements, the window:

- receives an incoming event
- executes DATA step code against the data in the event
- returns an output event

All fields in the input window are seen as variables by the DATA step.

Use a SET statement to receive the event and populate the DATA step variables. Use an OUTPUT statement to create an Upsert event, which is returned to the Procedural window. Both the SET and OUTPUT statements reference the event stream processing libref, which requires the `sasioesp` load module to be in the SAS search path.

Configuration

When you configure a model that contains a Procedural window that executes DATA step code, the project must contain the `<ds-initializer>` element:

```
<ds-initialize
                sas-log-location='@SAS_LOG_DIR@'
                sas-connection-key='5555'
                sas-command='sas -path @DFESP_HOME@/lib'
  />
```

- The `sas-log-location` is optional. If you do not specify it, the SAS log is placed in the directory where the event stream processing server was started.
- The `sas-connection-key` is optional. This key is used as the shared memory and semaphore key to communicate with Base SAS. It is a system-level resource (like a port) and needs to be unique per event stream processing server executing on the system. When there is only one event stream processing server running on the system, specify the default value of 5555.
- The `sas-command` starts a SAS session. It requires the **`-path`** option in order to find the access engine.

Within the Procedural window itself, specify the `<ds-external>` element as follows:

```
<ds-external source='request'
        trace='false'
        code-file='@SAS_SOURCE_DIR@/score.sas'
        connection-timeout='5'
        max-string-length='32'
  />
```

- The `source` attribute designates the Source window to which the remaining attributes apply.
- The `trace` flag turns on output to the SAS log. Use this flag only during the model development phase with small amounts of test data.
- The `code-file` attribute identifies which SAS program executes on events that arrive from the Source window.
- The `connection-timeout` is measured in seconds. The default value is 60 seconds. Consider increasing the value under the following circumstances:
  - when your SAS code is complex
  - when your code takes a long time to compile
  - when Base SAS performs extensive one time initialization, such as loading hash tables
- The `max-string-length` attribute communicates to Base SAS the maximum length of any string sent in an event from SAS Event Stream Processing to Base SAS.

Referencing in a DATA Step

Reference SAS Event Stream Processing in a DATA step as follows:

```
data esp.output;
     set esp.input;
     score = a * ranuni(104) + b;
  run;
```

- The DATA statement must designate esp.output as the output data set. When an observation is written to that data set, it actually is returned to the Procedural window as an Upsert event.
- The SET statement waits for the arrival of an event, and moves event data into DATA step variables.

Supported Data Types

The following mapping of event stream processor to DATA step data types is supported:

Event Stream Processor Data Type	DATA Step Data Type
ESP_INT32	Numeric variable. ESP NULL values map to SAS missing values, and vice versa.
ESP_INT64	Numeric variable. ESP NULL values map to SAS missing values, and vice versa.
ESP_DOUBLE	Numeric variable. ESP NULL values map to SAS missing values, and vice versa.
ESP_TIMESTAMP ESP DATETIME	Numeric variable whose value is the number of seconds since Jan 1, 1960. ESP NULL values map to SAS missing values and vice versa. .

Event Stream Processor Data Type	DATA Step Data Type
ESP_UTF8STR	Character Variable. SAS Character variables are trimmed before being returned to SAS Event Stream Processing.
ESP_MONEY	Not supported.

Known Limitations

- Currently this functionality is supported only on Linux platforms
- Some DATA step statements and options do not make sense when you use them in a real-time event processing context. For example, you should not use the END= option in the SET statement. In a real-time system, it is not known whether there are more records to come.
- The Procedural window uses shared memory and system semaphores to communicate with Base SAS. These are system wide resources, similar to sockets. Therefore, event stream processing servers that run on the same system cannot use the same set of keys to communicate with Base SAS. You can use the `sas-connection-key` attribute on the `ds-initialize` element to alter the starting key for one of the event stream processing servers.
- SAS Event Stream Processing supports mixed-case field names. Base SAS does not.
- SAS Event Stream Processing supports varying length strings. The SAS access engine interface does not. Use the `max-string-length` attribute on the Procedural window's `ds-external` element to declare the length of the maximum expected string value that is sent to Base SAS.

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,sqrtValue: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. For more information, see the *SAS Micro Analytic Service: Programming and Administration Guide*.

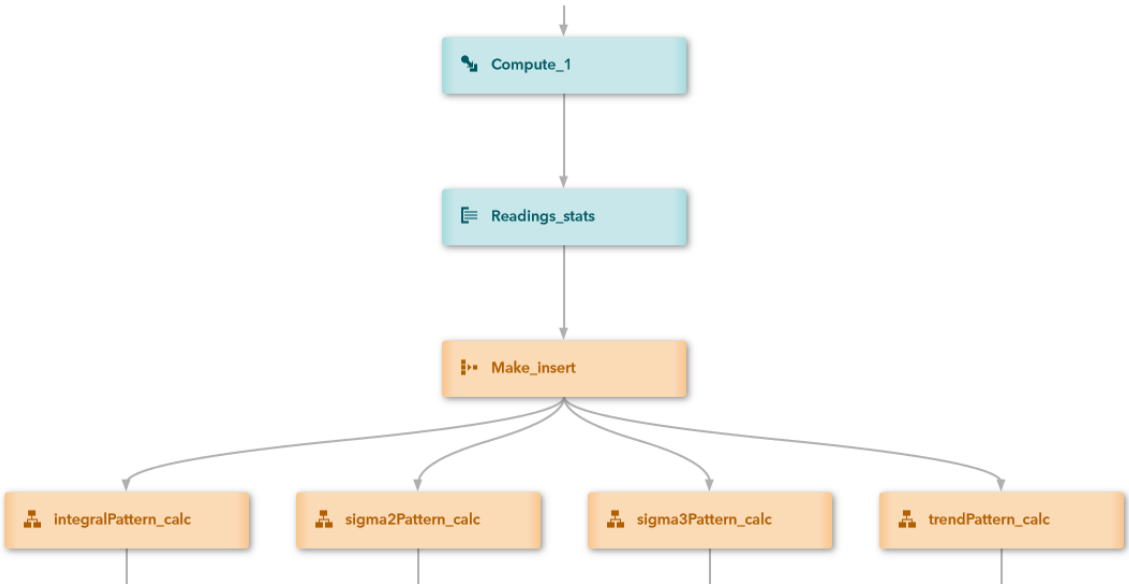
Using Remove State Windows

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 Dictionary* .

Consider the following model:

Figure 7 *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.

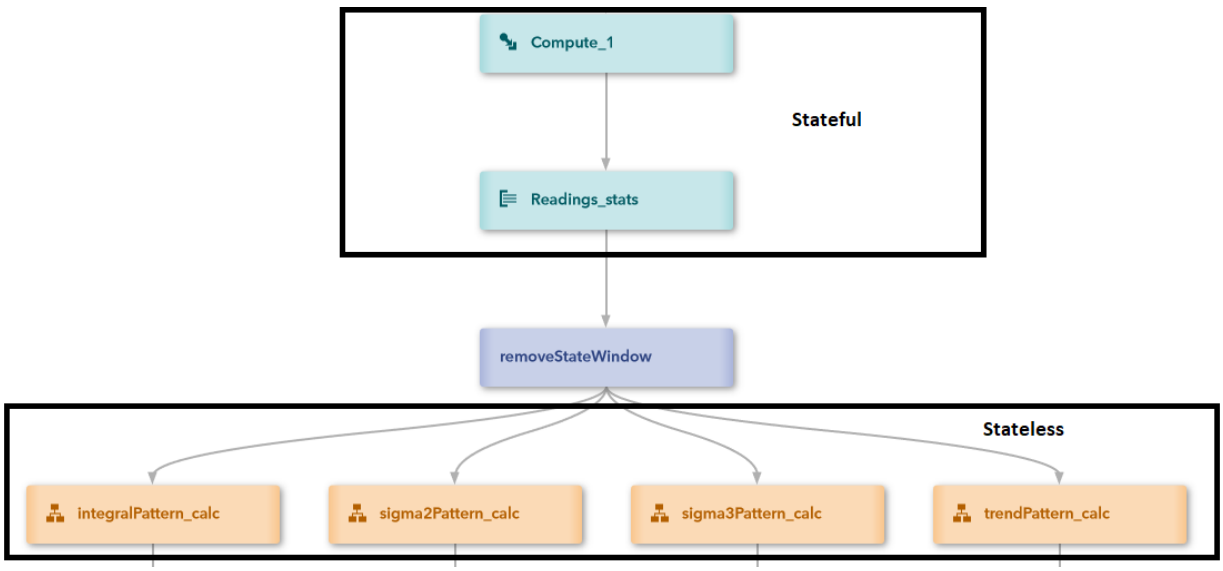
Compute_1 streams augmented events into Readings_stats, an Aggregate window. Readings_stats groups events by a location and calculates values such as sum, average, standard deviation, and so on, for the group. Every time that a new event for a specific location streams into Reading_stats, an Update event 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 8 *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>
```

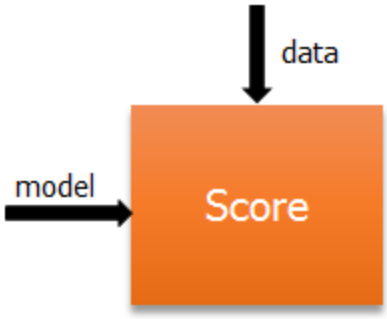
1 The `eventNumber` is inserted by the Remove State window into each event that passes through.

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 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.)



For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

Using Text Category Windows

Use a Text Category window to categorize a text field in incoming events. The text field can generate zero or more categories, with scores. Text Category windows are insert-only, so they require an index of `pi_EMPTY`.

For information about the XML elements associated with a Text Category window, see "window-textcategory" in *SAS Event Stream Processing: XML Language Dictionary* .

Note: Without SAS Contextual Analysis, you do not have the MCO file that is required to initialize a Text Category window. For more information about SAS Contextual Analysis, see the *SAS Contextual Analysis: User's Guide*.

Using Text Context Windows

Use a Text Context window to abstract classified terms from an unstructured string field. Use this window to analyze a string field from an event's input to find classified terms. Events generated from the terms can be analyzed by other window types. For example, a Pattern window could follow a Text Context window to look for tweet patterns of interest. Text Context windows are insert-only, so they require an index of `pi_EMPTY`.

For information about the XML elements associated with Text Context windows, see "window-textcontext" in *SAS Event Stream Processing: XML Language Dictionary* .

Note: Without SAS Contextual Analysis, you do not have the language initialization files that are required to initialize a Text Context window. For more information about SAS Contextual Analysis, see the *SAS Contextual Analysis: User's Guide*.

Using Text Sentiment Windows

Use a Text Sentiment window to determine the sentiment of text in the specified incoming text field and the probability of its occurrence. The sentiment value is "positive," "neutral," or "negative." The probability is a value between 0 and 1. Text Sentiment windows are insert-only, so they require the index type `pi_EMPTY`.

For information about the XML elements associated with a Text Sentiment window, see "window-textsentiment" in *SAS Event Stream Processing: XML Language Dictionary* .

Note: Without SAS Sentiment Analysis Studio, you do not have the SAM file that is required to initialize a Text Sentiment window. For more information about SAS Sentiment Analysis Studio, see the *SAS Sentiment Analysis Studio: User's Guide*.

The following Source window reads a CSV file named text_sentiment_data through a file and socket connector.

```
<window-source name='sourceWindow_01'>
    <schema-string>ID*:int32,tstamp:date,msg:string</schema-string>
       <connectors>
        <connector class='fs'>
         <properties>
          <property name='type'>pub</property>
          <property name='fstype'>csv</property>
          <property name='fsname'>text_sentiment_data.csv</property>
          <property name='dateformat'>%Y-%m-%d %H:%M:%S</property>
         </properties>
        </connector>
       </connectors>
</window-source>
```

The data in the CSV file, which conforms to the schema specified by the Source window, looks something like this:

```
i    n    1    9/7/2010 16:09    I love my pickup truck
```

The Text Sentiment window specifies an empty primary index, the SAM file path, and the input string field or document to analyze. Change the samFile string to point to a specific SAM file.

```
<window-textsentiment name='textSentimentWindow' sam-file=samFile.sam' text-field='msg' index='pi_EMPTY'/>
```

Remember, you must have SAS Sentiment Analysis Studio to initialize a Text Sentiment window.

Place a copy window after the Text Sentiment window so that it can hold text sentiment events using a retention policy. Here, the retention policy is set to a sliding window of five minutes.

```
<window-copy name='copyWindow_01' index='pi_RBTREE'>
   <retention type='bytime_sliding'>5 minutes</retention>
</window-copy>
```

Here are the edges that connect the windows.

```
<edges>
      <edge source='sourceWindow_01' target='textSentimentWindow'/>
      <edge source='textSentimentWindow' target='copyWindow_01'/>
</edges>
```

Using Text Topic Windows

Overview

Text topic windows run Text Topic models on events to score and identify themes in streaming text. Text Topic models are generated by SAS Visual Text Analytics or SAS Visual Data Mining and Machine Learning. Text Topic windows receive and process text from documents as string fields. Text Topic models enter a text topic window through an analytic store file. Text Topic windows are insert-only, so they require the index type `pi_EMPTY`.

For information about the XML elements associated with Text Topic windows, see “window-texttopic” in *SAS Event Stream Processing: XML Language Dictionary* .

For more information about creating text mining models and saving them to analytic store files, see *SAS Visual Data Mining and Machine Learning: Data Mining and Machine Learning Procedures*.

Example of Text Topic Window

Consider the following example.

A single continuous query contains a Source window (`sourceWindow_01`) and a Text Topic window (`textTopicWindow`).

The Source window reads a file containing text that needs to be analyzed:

```
<window-source name="sourceWindow_01"
        pubsub="true" collapse-updates="true"
        index="pi_EMPTY" insert-only='true'>
          <schema>
            <fields>
              <field name='did' type='string' key='true'/>
              <field name='text' type='string'/>
            </fields>
          </schema>
          <connectors>
            <connector name="pub" class="fs">
              <properties>
                <property name="type"><![CDATA[pub]]></property>
                <property name="fstype"><![CDATA[csv]]></property>
                <property name="fsname"><![CDATA[input/input.csv]]></property>
                <property name='dateformat'>%Y-%m-%d %H:%M:%S</property>
              </properties>
            </connector>
          </connectors>
        </window-source>
```

The Text Topic window receives the streaming events and analyzes the text according to the model in the analytic store file specified at the `window-texttopic` level:

```
<window-texttopic name="textTopicWindow"
                          index="pi_EMPTY"
                          pubsub="true"
                          astore-file="../common/binary/astore"
                          ta-path=".../tools/textAnalytics/SASFoundation/9.4/misc/tktg"
                          text-field="text"
                          include-topic-name="true">
        </window-texttopic>
```

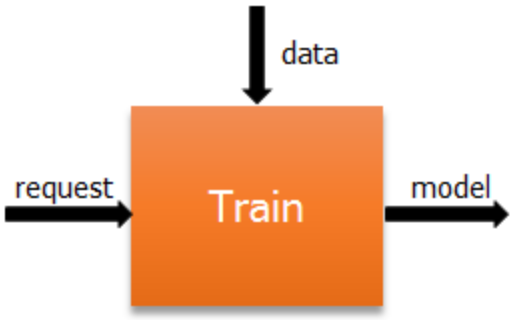
Edges connect the source window to the Text Topic window at the end of the continuous query:

```
<edges>
        <edge source="sourceWindow_01" target="textTopicWindow"/>
      </edges>
```

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.

For more information, see *SAS Event Stream Processing: Using Streaming Analytics*.

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 Dictionary* .

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

Table 10 *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'
/>
```

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 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 Dictionary* .

All input windows to a Union window must have the same schema. The default value of the strict flag is **`true`**, which means that the key merge from each window must semantically merge cleanly. In this case, you cannot send an Insert event for the same key using two separate input windows of the Union window.

Setting the strict flag set to **`false`** loosens the union criteria by replacing all incoming Inserts with Upserts. All incoming Deletes are replaced with safe Deletes. In this case, Deletes of a non-existent key fail without generating an error.

Understanding Retention

Any Source or Copy window can set a retention policy. A window's retention policy governs how it introduces Deletes into the event stream. These Deletes work their way along the data flow, recomputing the model along the way. Internally generated Deletes are flagged with a retention flag, and all further window operations that are based on this Delete are flagged.

Note: Under some circumstances when you map a MAS method to a Procedural 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.
volume-based	Retention is based on a specified number of records. When the volume increases beyond that specification, the oldest events are removed.

Both time and volume-based retention can occur in one of two variants:

Retention Variant	Description
sliding	Specifies a continuous process of deleting events. Think of the retention window sliding continuously. For a volume-based sliding window, when the specified threshold is hit, one delete is executed for each insert that comes in.
jumping	Specifies a window that completely clears its contents when a specified threshold value is hit. Think of a ten-minute jumping window as one that deletes its entire contents every 10 minutes.

A time-based retention mode called `bytime_jumping_lookback` enables you to set retention based on a specified unit of time. You configure it through two parameters:

- `unit` (**`minute`**, **`hour`**, **`day`**, **`week`**, **`month`**, or **`year`**)
- `value`, which specifies the retention period. Use a positive integer that represents a multiple of the specified `unit`.

When an event streams into a window, it is rounded up to the specified UNIT.

For example, suppose that the `unit` is **`month`** and the first event arrives at **`08-29-2017 17:21:00`**. The rounded up time becomes **`09-01-2017 00:00:00`**. That time becomes the end of the first retention period. All other retention periods are intervals of the look back `unit`.

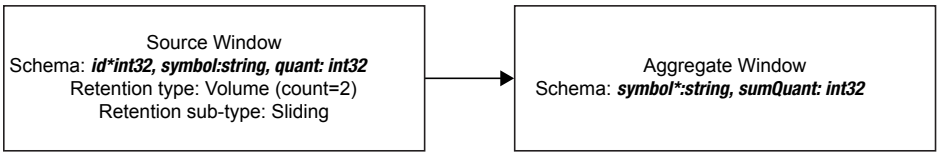
When the look back is 2, the next retention period would be **`[09-01-2017 00:00:00, 11-01-2017 00:00:00)`** and the next one after that would be: **`[11-01-2017 00:00:00, 01-01-2018 00:00:00)`**.

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

Retention Type and Variant	Use Case Examples
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 11 *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 12 *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

Primary Index Type	Algorithm	Storage	Advantages	Drawbacks
`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.

Using a Large Lookup Table with Persistence Across Restarts

The following example shows how to enable an efficient no-regenerate lookup join with the following properties:

- 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.

Suppose that you have a project named pr_01 that contains a continuous query named cq_01. The query contains two Source windows. Both of those windows are stateless, that is, they have indexes of `pi_EMPTY`.

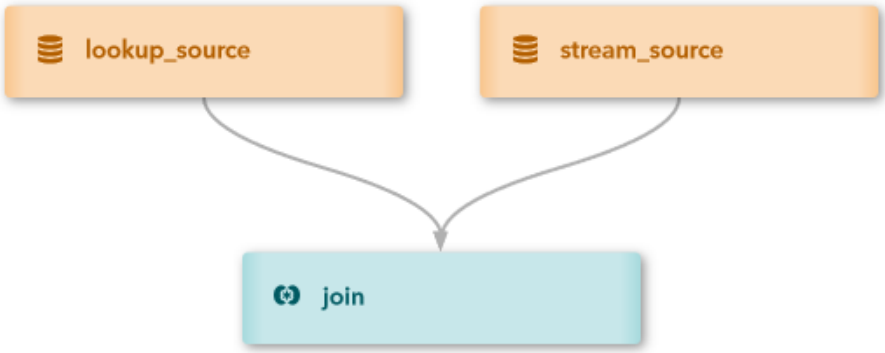
```
<window-source name="stream_source" index="pi_EMPTY" insert-only='true'>
               <schema>
                  <fields>
                     <field name='ID' type='int64' key='true'/>
                     <field name='matchID' type='int64'/>
                  </fields>
               </schema>
               <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>
```

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 9 *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

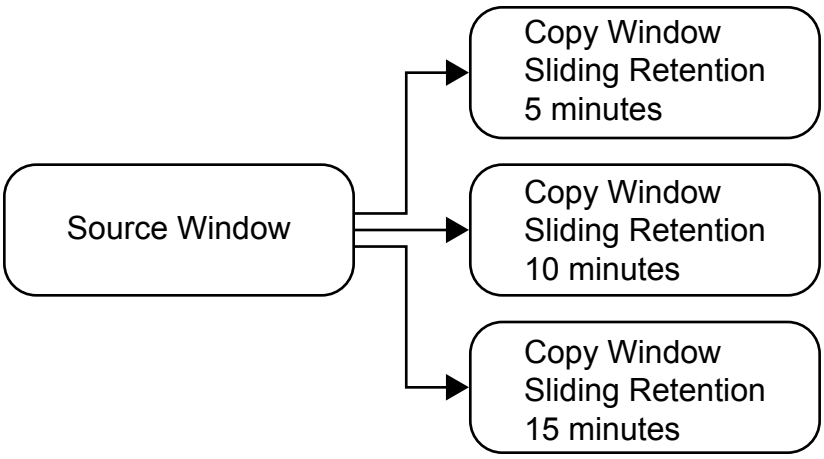
Any window that uses a primary index type `pi_EMPTY` is non-stateful or stateless. It acts as a pass-through for all incoming events. This index does not store events.

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 10 *Copy Windows*

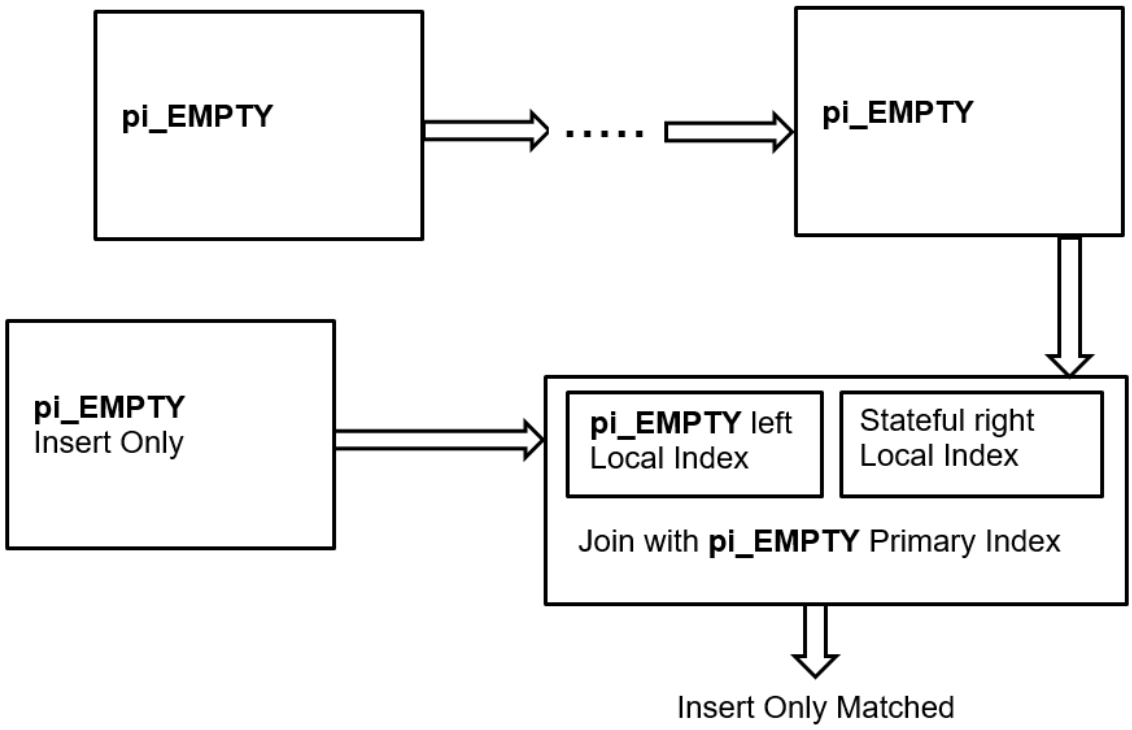


Using a Stateful Local Join Index to Resolve the State

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 11 *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.

Table 13 *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

Window	Index Restriction	Input Window Restriction	Opcodes Output
Compute	None	None	Produces only Inserts when the input window produces only Inserts
Copy	Use only stateful indexes; this requires that retention is set. See Note.	None	All
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
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.
Remove State	Use `pi_EMPTY` exclusively	None	Always produces Inserts
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

Window	Index Restriction	Input Window Restriction	Opcodes Output
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 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.

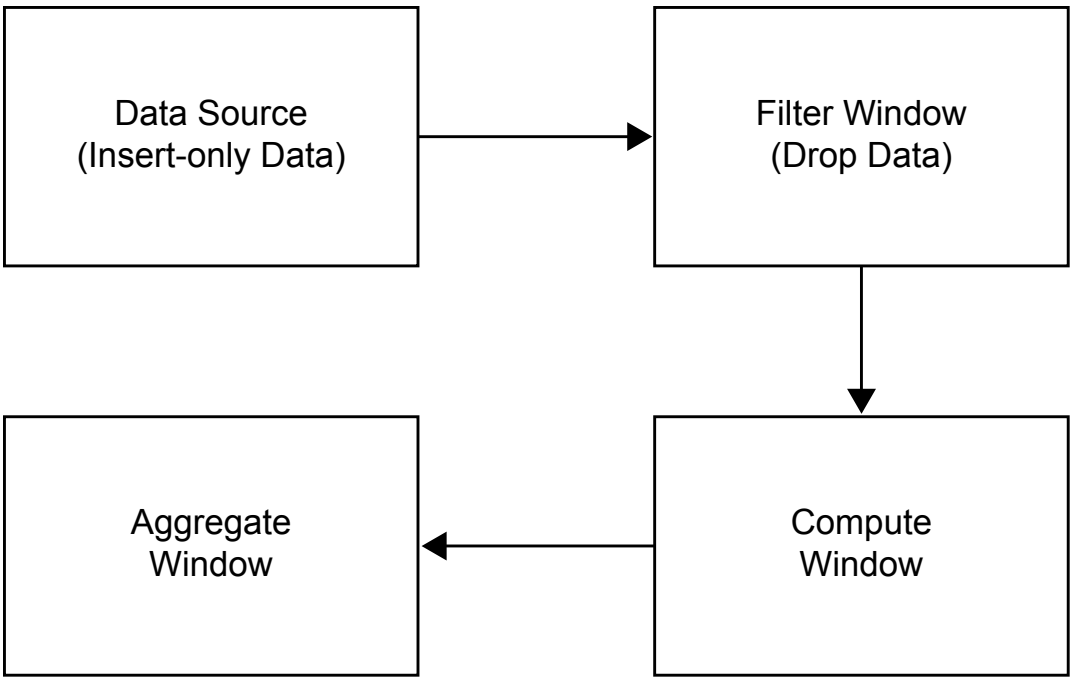
A mixed model has stateless and stateful parts. Often it is possible to separate the parts into a stateless front end and a stateful back end.

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 12 *Event Stream Processing Model with Insert-Only Data*



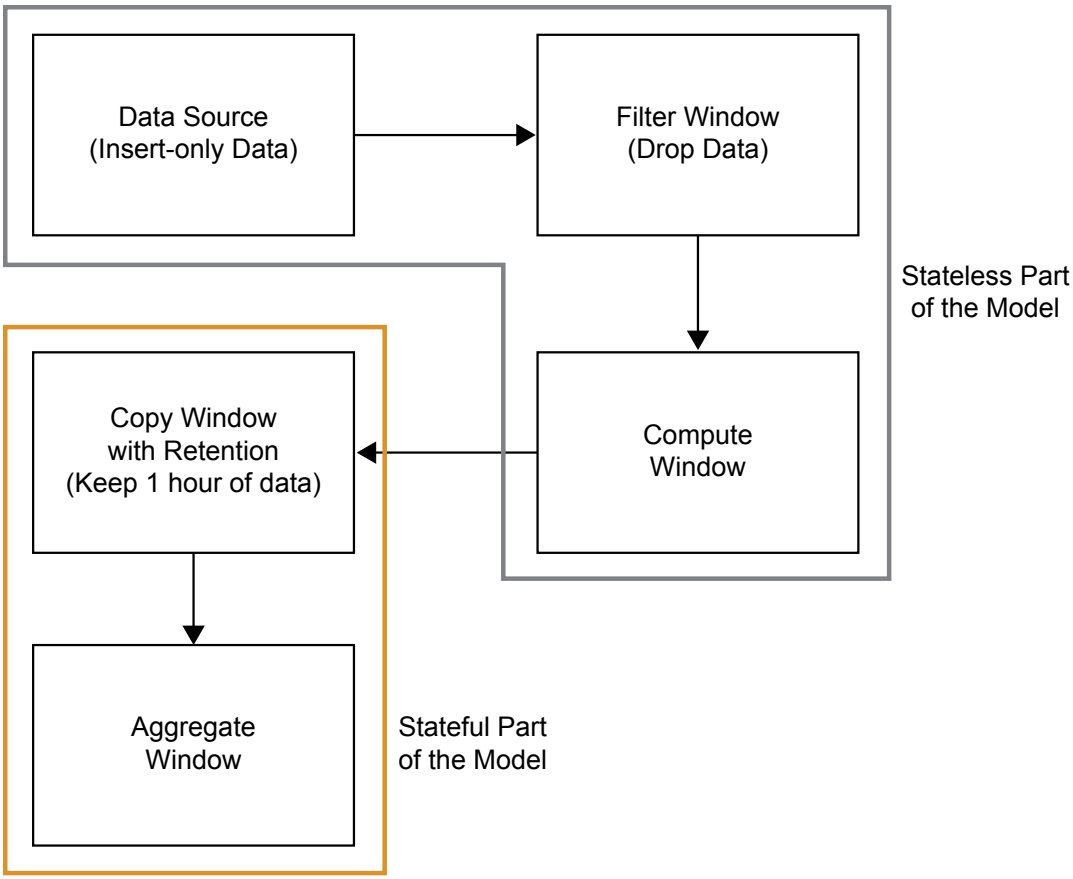
Here the data source is purely through Inserts. Therefore, the model can be made stateless by using an index type of **`pi_EMPTY`**. The filter receives inserts from the source, and drops some of them based on the filter criteria, so it produces a set of inserts as output. Thus, the filter can be made 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 13 *Modified Event Stream Processing Model with Stateless and Stateful Parts*



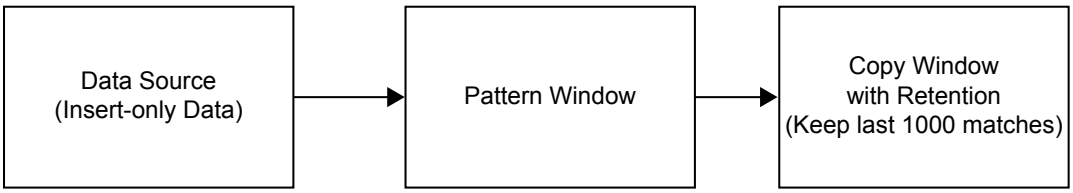
The stateful part of the model is accurately computed, based on the rolling window of input data. This model is bounded in memory.

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 14 *Event Stream Processing Model with Copy with Retention*



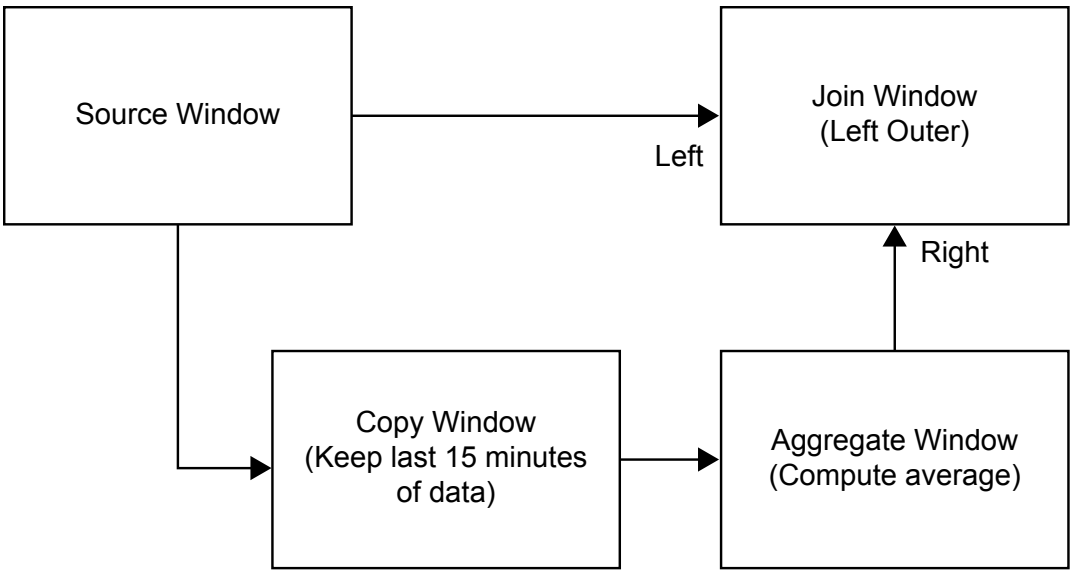
In cases like these, it is more likely that the Copy window is queried from the outside using adapters, or publish/subcribe clients. The Copy window might also feed other sections of the model.

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 15 *Event Stream Processing Model Using Advanced Features*



To control the aggregate window:

- Put retention before it (the Copy window).
- Group it by symbol (which is bounded), and use the additive aggregation function average (`ESP_aAve`), which does not need to store each event for its computation.

The Join window can be problematic. Ordinarily you think of a Join window as stateful. A join retains data for its fact window or dimension window and performs matches. In this case, you want a special, but frequently occurring behavior. When input comes in, pause it at the join until the aggregate corresponding to the input is processed. Then link the two together, and pass the augmented insert out.

To process input in this way:

1. Make the join a left outer join, with the source feed the left window, and the aggregate feeds the right window.
2. Set Tagged Token data flow model on for the projects. This turns on a special feature that causes a fork of a single event to wait for both sides of the fork to rejoin before generating an output event.
3. Set the index type of the join to **`pi_EMPTY`**, making the join stateless. A stateless left outer join does not create a local copy of the left driving window (FACT window). It does not keep any stored results of the join. However, there is always a reference-counted copy the lookup window. In the case of a left outer join, this is the right window. The lookup window is controlled by retention in this case, so it is bounded.
4. Ordinarily, a join, when the dimension window is changed, tries to find all matching events in the fact window and then issue updates for those joined matches. You do not want this behavior, because you are matching events in lock step. Further, it is simply not possible because you do not store the fact data. To prevent this regeneration on each dimension window change, set the **`no-regenerates`** option on the Join window.

In this way you create a fast, lightweight join. This join stores only the lookup side, and produces a stream of inserts on each inserted fact event. A stateless join is possible for left and right outer joins.

The following XML code implements this model.

```
<engine port='52525' dateformat='%d/%b/%Y:%H:%M:%S'>
    <projects>
        <project name='trades_proj' pubsub='auto'
                 use-tagged-token='true' threads='4'>
            <contqueries>
                <contquery name='trades_cq'>

                    <windows>
                        <window-source name='Trades'
                                       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='bycount_sliding'>5</retention>
                        </window-copy>

                        <window-aggregate name='RunTotal'>
                            <schema>
                                <fields>
                                    <field name='tradeID' type='string'/>
                                    <field name='security' type='string' key='true'/>
                                    <field name='quantityTotal' type='double'/>
                                </fields>
                            </schema>
                            <output>
                                <field-expr>ESP_aLast(tradeID)</field-expr>
                                <field-expr>ESP_aSum(quantity)</field-expr>
                            </output>
                        </window-aggregate>

                        <window-join name='JoinTotal'
                                     index='pi_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>
  </projects>
</engine>
```

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

```
// the uMyFunction wrapper:
     // every aggregation function must be wrapped like this.
    //
    int dfESPaggrfunc_uMyFunctionWrapper(dfESPexpEngine::exp_engine_t *e,
```

```
                                        dfESPexpEngine::exp_sym_value_t *returnval,
                                        int parmcount,
                                        dfESP_EXPsym_value_t **parms)
                                         {
        return dfESPaggrfunc_Wrapper((void *)my_aggrergation_function, e,
                                           returnval, parmcount, parms);
                                         }
```

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

```
// 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 parameters are: <callable name>, <function pointer>,
        // <num args>, <additive flag>, <additive flag for insert only>,
        // <description>
        uFuncts.push_back(new aggr_function_t("USER_uSum_add",
           (void *)dfESPaggrfunc_uMyFunctionWrapper, "a", true,
            true, "description"));
    }
```

Adjust the number of arguments, additive flags, and the description field accordingly. The sample code **`$DFESP_HOME/examples/cxx/aggregate_userdef/src/functions.cpp`** provides two complete examples. The makefile distributed with the sample code produces a shared library in the **`aggregate_userdef/plugins`** directory. Copy this plug-in to **`$DFESP_HOME/lib`** and name it `libdfxesp_udafD-major.minor`.

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.

Splitting Generated Events across Output Slots

Overview

All window types can register a splitter function or expression to determine what output slot or slots should be used for a newly generated event. This enables you to send generated events across a set of output slots.

Most windows send all generated events out of output slot 0 to zero or more downstream windows. For this reason, it is not standard for most models to use splitters. Using window splitters can be more efficient than

using Filter windows off a single output slot. This is especially true, for example, when you are performing an alpha-split across a set of trades or a similar task.

When adding 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. If the slot number is -1, the downstream window receives all the data produced by the parent window regardless of the splitter function.

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 XML

Here is an example with one Source window, one Compute window, and three Copy windows. The Compute window includes a user-defined function as a splitter to determine what slot an event should go to. Each slot directs to a different Copy window.

Here is the Source window:

```
<window-source name='sourceWindow' index='pi_RBTREE'>
       <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'>
         <properties>
          <property name='type'>pub</property>
          <property name='fstype'>csv</property>
          <property name='fsname'>input.csv</property>
          <property name='blocksize'>1</property>
          <property name='transactional'>true</property>
         </properties>
        </connector>
       </connectors>
      </window-source>
```

The Source window uses a file and socket publisher connector to receive input events from a CSV file named input.csv.

```
i    n    1    IBM    121.48
i    n    2    AMZN    1593.41
u    n    2    AMZN    1588.2
p    n    3    APPL    179.55
p    n    3    APPL    198
i    n    5    GOOGL    1094.58
d    n    1    IBM    120.5
i    n    11    FB    138.68
u    n    11    FB    137.5
```

It streams those events to a Compute window:

```
<edge source='sourceWindow' target='computeWindow'/>
```

The Compute window includes a user-defined function as a splitter. The function calculates a slot number to determine what Copy window should receive the incoming event:

```
<window-compute name='computeWindow' collapse-updates='true'>
```

```
        <splitter-expr>
         <expr-initialize>
          <udfs>
           <udf name='udf1' type='int32'>
            <![CDATA[private integer p
                          p = parameter(1);
                          return p%2]]>
           </udf>
          </udfs>
         </expr-initialize>
         <expression>udf1(ID)</expression>
        </splitter-expr>
        <expr-initialize>
         <initializer type='int32'>
          <![CDATA[integer counter
                     counter=0]]>
         </initializer>
        </expr-initialize>
        <schema>
         <fields>
          <field name='ID' type='int32' key='true'/>
          <field name='counter' type='int32'/>
          <field name='symbol' type='string'/>
          <field name='price' type='double'/>
         </fields>
        </schema>
        <output>
         <field-expr>counter=counter+1 return counter</field-expr>
         <field-expr>symbol</field-expr>
         <field-expr>price</field-expr>
        </output>
       </window-compute>
```

Here are the edges between the Compute and Copy windows, specifying the slot numbers of the Compute window where the Copy windows receive their input events.

```
<edge source='computeWindow' slot='0' target='computeWindowSlot_01'/>
      <edge source='computeWindow' slot='1' target='computeWindowSlot_02'/>
      <edge source='computeWindow' slot='-1' target='computeWindowSlot_03'/>

<window-copy name='computeWindowSlot_01'/>
      <window-copy name='computeWindowSlot_02'/>
      <window-copy name='computeWindowSlot_03'/>
```

After streaming the incoming events, here are the events processed by computeWindowSlot_01:

```
I,N, 2,2,AMZN,1593.410000
UB,N, 2,3,AMZN,1588.200000
D,N, 2,2,AMZN,1593.410000
```

Here are the events processed by computeWindowSlot_02:

```
I,N, 1,1,IBM,121.480000
I,N, 3,4,APPL,179.550000
UB,N, 3,5,APPL,198.000000
D,N, 3,4,APPL,179.550000
I,N, 5,6,GOOGL,1094.580000
D,N, 1,1,IBM,121.480000
I,N, 11,7,FB,138.680000
UB,N, 11,8,FB,137.500000
```

```
D,N, 11,7,FB,138.680000
```

Here are the events processed by computeWindowSlot_03:

```
I,N, 1,1,IBM,121.480000
I,N, 2,2,AMZN,1593.410000
UB,N, 2,3,AMZN,1588.200000
D,N, 2,2,AMZN,1593.410000
I,N, 3,4,APPL,179.550000
UB,N, 3,5,APPL,198.000000
D,N, 3,4,APPL,179.550000
I,N, 5,6,GOOGL,1094.580000
D,N, 1,1,IBM,121.480000
I,N, 11,7,FB,138.680000
UB,N, 11,8,FB,137.500000
D,N, 11,7,FB,138.680000
```

Splitter Functions in C++

Here is a prototype for a C++ splitter function.

```
size_t splitterFunction(dfESPschema *outputSchema, dfESPeventPtr nev,
     dfESPeventPtr oev);
```

This splitter function receives the schema of the events supplied, the new and old event (only non-null for update block), and it returns a slot number.

Here is how you use the splitter for the Source window (`sw_01`) to split events across three Copy windows: `cw_01, cw_02, cw_03`.

```
sw_01->setSplitter(splitterFunction);
cq_01->addEdge(sw_01, 0,  cw_01);
cq_01->addEdge(sw_01, 1,  cw_02);
cq_01->addEdge(sw_01, -1, cw_03);
```

The `dfESPwindow::setSplitter()` member function is used to set the user-defined splitter function for the Source window. The `dfESPcontquery::addEdge()` member function is used to connect the Copy windows to different output slots of the Source window.

When no splitter function is registered with the parent window, the slots specified are ignored, and each child window receives all events produced by the parent window.

Note: Do not write a splitter function that randomly distributes incoming records. Also, do not write a splitter function that relies on a field in the event that might change. The change might cause the updated event to generate a different slot value than what was produced prior to the update. This can cause an Insert to follow one path and a subsequent Update to follow a different path. This generates inconsistent results, and creates indices in the window that are not valid.

Splitter Expressions in C++

When you define splitter expressions, you do not need to write the function to determine and return the desired slot number. Instead, the registered expression does this using the splitter expression engine. Applying expressions to the previous example would look as follows, assuming that you split on the field name "splitField", which is an integer:

```
sw_01->setSplitter("splitField%2");
cq_01->addEdge(sw_01, 0,  cw_01);
cq_01->addEdge(sw_01, 1,  cw_02);
cq_01->addEdge(sw_01, -1, cw_03);
```

Here, the `dfESPwindow::setSplitter()` member function is used to set the splitter expression for the Source window. Using splitter expressions rather than functions can lead to slower performance because of the overhead of expression parsing and handling. Most of the time you should not notice differences in performance.

`dfESPwindow::setSplitter()` has two additional optional parameters with defaults set to NULL.

- `initExp` enables you to specify an initialization expression for the expression engine used for this window's splitter.
- `initRetType` enables you to specify a return `datavar` value in those cases when you want to pass state from the initialization expression to the C++ application thread that makes the call. Most initialization expressions do not use return values from the initialization.

This initialization message enables you to specify some setup state, perhaps variable declarations and initialization, that you can use later in the splitter expression processing.

The full syntax for this call is as follows:

```
dfESPdatavarPtr setSplitter(const char* splitterExp, const char*
                            initExp=NULL, dfESPdatavar::dfESPdatatype
                            initRetType=dfESPdatavar::ESP_NULL);
```

You can find an example of window output splitter initialization in `splitter_with_initexp` in **`$DFESP_HOME/examples/cxx`**. The example uses the following `setSplitter` call where the initialize declares and sets an expression engine variable to 1:

```
(void)sw_01->setSplitter("counter=counter+1; return counter%2",
                                    "integer counter\r\ncounter=1");
```

For each new event the initialize increments and mods the counter so that events rotate between slots 0 and 1.

Marking Events as Partial-Update on Publish

Overview

In most cases, events are published into an engine with all fields available. Some of the field values might be null. Events with Delete opcodes require only the key fields to be non-null.

There are times when only the key fields and the fields being updated are desired or available for event updates. This is typical for financial feeds. For example, a broker might want to update the price or quantity of an outstanding order. You can update selected fields by marking the event as partial-update (rather than normal).

When you mark events as partial-update, you provide values only for the key fields and for fields that are being updated. In this case, the fields that are not updated are marked as data type `dfESPdatavar::ESP_LOOKUP`. This marking tells SAS Event Stream Processing to match key fields of an event retained in the system with the current event and not to update the current event's fields.

In order for a published event to be tagged as a partial-update, the event must contain all non-null key fields that match an existing event in the Source window. Partial updates are applied to Source windows only.

When using transactional event blocks that include partial events, be careful that all partial updates are for key fields that are already in the Source window. You cannot include the insert of the key values with an update to the key values in a single event block with transactional properties. This attempt fails and is logged because transactional event blocks are treated atomically. All operations in that block are checked against an existing window state before the transactional block is applied as a whole.

Publishing Partial Events into a Source Window

Consider these three points when you publish partial events into a Source window.

- In order to construct the partial event, you must represent all the fields in the event. Specify either the field type and value or a placeholder field that indicates that the field value and type are missing. In this way, the

existing field value for this key field combination remains for the updated event. These field values and types can be provided as `datavars` to build the event. Alternatively, they can be provided as a comma-separated value (CSV) string.

If you use CSV strings, then use '^U' (such as, control-U, decimal value 21) to specify that the field is a placeholder field and should not be updated. On the other hand, if you use `datavars` to represent individual fields, then those fully specified fields should be valid. Enter them as `datavars` with values (non-null or null). Specify the placeholder fields as empty `datavars` of type `dfESPdatavar::ESP_LOOKUP`.

- No matter what form you use to represent the field values and types, the representation should be included in a call for the partial update to be published. In addition to the fields, use a flag to indicate whether the record is a normal or partial update. If you specify partial update, then the event must be an Update or an Upsert that is resolved to an Update. Using partial-update fields makes sense only in the context of updating an existing or retained Source window event. This is why the opcode for the event must resolve to Update. If it does not resolve to Update, an event merge error is generated.

  If you use an event constructor to generate this binary event from a CSV string, then the beginning of that CSV string contains "`u,p`" to show that this is a partial-update. If instead, you use `event->buildEvent()` to create this partial update event, then you need to specify the event flag parameter as `dfESPeventcodes::ef_PARTIALUPDATE` and the event opcode parameter as `dfESPeventcodes::eo_UPDATE`.

- One or more events are pushed onto a vector and then that vector is used to create the event block. The event block is then published into a Source window. For performance reasons, each event block usually contains more than a single event. When you create the event block, you must specify the type of event block as transactional or atomic using `dfESPeventblock::ebt_TRANS` or as normal using `dfESPeventblock::ebt_NORMAL`.

  Do not use transactional blocks with partial updates. Such usage treats all events in the event block as atomic. If the original Insert for the event is in the same event block as a partial Update, then it fails. The events in the event block are resolved against the window index before the event block is applied atomically. Use normal event blocks when you perform partial Updates.

Examples

Here are some sample code fragments for the variations on the three points described in the previous section.

Create a partial Update `datavar` and push it onto the `datavar` vector.

```
// Create an empty partial-update datavar.
dfESPdatavar* dvp  = new dfESPdatavar(dfESPdatavar::ESP_LOOKUP);
// Push partial-update datavar onto the vector in the appropriate
// location.
// Other partial-update datavars might also be allocated and pushed to the
// vector of datavars as required.
dvVECT.push_back(dvp); // this would be done for each field in the update
     event
```

Create a partial Update using partial-update and normal `datavars` pushed onto that vector.

```
// Using the datavar vector partially defined above and schema,
// create event.
dfESPeventPtr eventPtr = new dfESPevent();
eventPtr->buildEvent(schemaPtr, dvVECT, dfESPeventcodes::eo_UPDATE,
dfESPeventcodes::ef_PARTIALUPDATE);
```

Define a partial update event using CSV fields where '`^U`' values represent partial-update fields. Here you are explicitly showing '`^U`'. However, in actual text, you might see the character representation of `Ctrl-U` because individual editors show control characters in different ways.

Here, the event is an Update (due to 'u'), which is partial-update (due to 'p'), key value is 44001, "ibm" is the instrument that did not change. The instrument is included in the field. The price is 100.23, which might have changed, and 3000 is the quantity, which might have changed, so the last three of the fields are not updated.

```
p = new dfESPevent(schema_01,
(char *)"u,p,44001,ibm,100.23,3000,^U,^U,^U");
```

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. For more information about online training, see "Overview" in *SAS Event Stream Processing: Using Streaming Analytics*.

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. For more information about offline training, see "Overview" in *SAS Event Stream Processing: Using Streaming Analytics*.

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 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.
-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)`

Functional Window and Notification Window Support Functions

Functions for Event Stream Processing

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 14 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 15 Arguments to ESPCONFIGVALUE

Argument	Description
argument	Specifies a configuration value. For more information about configuration values, see "Configuring the ESP Server" in *SAS Event Stream Processing: Using the ESP Server*.

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
```

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
```

PROJECTNAME

Returns the name of the project containing the current event.

projectName()

```
projectName()=project_1
projectName()=myproject
```

General Functions

ABS

Returns the absolute floating-point value of the supplied argument.

abs(*argument*)

Table 16 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 17 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 18 *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 19 *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 20 *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 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 22 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. ■ 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 23 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 24 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 25 *Arguments to CONCAT*

Argument	Description
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.

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 26 *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 27 *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 28 *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 29 *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 30 *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`**. ■ 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 31 *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 32 Arguments to FLOOR

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.

```
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 33 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 34 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 only supports integers.

increment(*argument*)

Table 35 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 36 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 37 Arguments to IFNEXT

Argument	Description
argument	Specifies a Boolean expression.

Argument	Description
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 38 *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 39 *Arguments to INDEX*

Argument	Description
index	Specifies an integer or a function that returns an integer.

Argument	Description
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 40 *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 41 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 42 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. ■ 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 43 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 44 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 45 *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
```

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 46 *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 47 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 48 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 49 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 50 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 51 *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 52 *Arguments to MAPVALUE*

Argument	Description
argument	Specifies a delimited string of name-value pairs.
outerdelimiter, *innerdelimiterdelimiter*	Specifies characters used as delimiters.
name	Specifies the name in the name-value pair specified in *argument*.
#reference	Specifies a reference to a function context.

```
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 53 *Arguments to MAPVALUES*

Argument	Description
argument	Specifies a delimited string of name-value pairs.
outerdelimiter, *innerdelimiterdelimiter*	Specifies characters used as delimiters.
name1, ...*nameN*	Specifies the name in the name-value pair specified in *argument*.
#reference	Specifies a reference to a function context.

```
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 54 *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. ■ 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 55 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 56 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 57 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 58 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 59 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 60 *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
```

NOT

Returns true when the Boolean value of the argument is false. Otherwise, returns false.

not(*argument*)

Table 61 *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 62 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`**. ■ 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 63 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 64 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 65 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 66 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 67 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 68 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 69 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 70 *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 71 *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 72 *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 73 *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 74 *Arguments to RGXREPLACEALL*

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.

```
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 75 *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 76 *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 77 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 78 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 79 *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. ■ 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 80 *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 81 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 82 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 83 Arguments to SUBSTRING

Argument	Description
argument	Specifies a string.
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 84 *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 85 *Arguments to SUBSTRINGBEFORE*

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.

```
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 86 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 87 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 88 *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 89 *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 90 *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 91 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 92 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 93 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 94 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 95 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 96 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 97 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 98 Arguments to TIMEPARSE

Argument	Description
time	Specifies a time specification or a function that returns a time specification.
format	Specifies a time format that is supported by the UNIX `strftime` 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 99 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 100 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 101 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 *format*, the system default time format is used.

```
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 102 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 103 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 104 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 105 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 106 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 107 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 108 *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 109 *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 110 *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
```