THE POWER TO KNOW.

SAS® Event Stream Processing 4.2: Programming Reference

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Event Stream Processing 4.2: Programming Reference*. Cary, NC: SAS Institute Inc.

SAS® Event Stream Processing 4.2: Programming Reference

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

September 2016

4.2-P1:espmdlgobj

Contents

SAS Event Stream Processing 4.2: Programming Reference

Audience

This book is written for experienced C++ programmers who build event stream processing applications using SAS Event Stream Processing. Open **`$DFESP_HOME/doc/html/index.html`** in a web browser to access the complete class and method documentation for C++ modeling objects.

Chapter 1

C++ Modeling Objects for SAS Event Stream Processing

Overview to the C++ Modeling API

The C++ Modeling API provides a set of classes with member functions for each of the modeling objects. These classes enable event stream processing application developers to build event stream processing engines as a stand-alone event stream processing server. Each engine contains one or more projects.

Alternatively, you can embed an event stream processing engine into the process space of either an existing application or a new application. In that case, the main application thread is focused on its own chores. It interacts with the embedded engine as needed. Each project in the engine has its own dedicated thread pool.

The following sections describe common C++ modeling objects. Open **`$DFESP_HOME/doc/html/index.html`** in a web browser to access the complete class and method documentation for C++ modeling objects.

Dictionary

dfESPengine

specifies the top-level container or manager of an event stream processing instance. An engine can be a stand-alone executable or embedded in a C++ application. Engines contain one or more projects. dfESPengine is a singleton class instance; any process can have at most one dfESPengine within it.

Syntax

DFESP_API static dfESPengine *dfESPengine::initialize(int argc, char *argv[],
dfESPstring *id*, pubsubSpec_t pubsub, dfESPLoggingLevel *logLevel*=dfESPLLInfo,
const char **logConfigFile*=NULL,
const char **licKeyFile*=NULL);

Required Arguments

id
specifies the engine ID

argc
argument count as passed into main

argv
argument vector as passed into main.

Accepts the following options:

- **`-t`** *textfile.name* to write output
- **`-b`** *badevent.name* to write events that failed to be applied to a window index
- **`-r`** *restore.path* to restore a previously persisted engine state.
- **`-h`** *http-pubsub-port* to specify a port for a restful publish/subscribe interface. Use this to access the server from Streamviewer.

pubsub
indicates whether to enable **`(pubsub_ENABLE(port_number))`** or disable **`(pubsub_DISABLE)`** publish/subscribe for the engine.

Optional Arguments

logLevel
the lower threshold for displayed log messages. The default value is **`dfESPLLInfo`**.

logConfigFile
a logging facility configuration file. The default is to configure logging to go to standard output.

licKeyFile
a fully qualified pathname to a license file. The default is **`$DFESP_HOME/etc/license/esp.lic`**.

Details

You can use the following method to tell an engine how to handle fatal errors that occur in a project.

static DFESP_API void dfESPengine::setProjectFatalDisposition (projectFatal_t *dispositionFlag*)

Set the *dispositionFlag* to one of the following values:

Value	Description
dfESPengine::projectFatal_EXIT_ENGINE	exit with the engine process
dfESPengine::projectFatal_EXIT_ENGINE_WITH_CORE	exit and generate a core file for debugging, and stop all processing
dfESPengine::projectFatal_EXIT_PROJECT	disconnect publish/subscribe, clean up all threads and memory, and remove the process from the engine while leaving the engine up and processing other projects

Example

The following example creates, starts, stops, and shuts down an engine.

```
// Create the engine container.
dfESPengine *engine;
engine = dfESPengine::initialize(argc, argv, "engine", pubsub_DISABLE);

// Create the rest of the model and then start the projects.
//     Creating the rest of the model is covered in the
//     subsequent sections.
engine->startProjects();

// The project is running in the background and you can begin
//     publishing event streams into the project for execution.

/* Now cleanup and shutdown gracefully */

// First, stop the projects.
engine->stopProjects();

// Next, shutdown the engine and its services (this frees all
//     the modeling resources from the engine down through
//     the windows).
engine->shutdown();
```

dfESPdatavar

represents a variable that can hold any data type that the event stream processing engine supports. It is essentially a C union of event stream processing data types.

Syntax

// construct a new NULL datavar type
DFESP_API dfESPdatavar(dfESPdatatype *data_type*);

// construct a new NULL datavar type from an existing datavar
DFESP_API dfESPdatavar(dfESPdatavar **dv*);

// construct a new NULL datavar type from binary data
DFESP_API dfESPdatavar(dfESPdatatype *data_type*, void **b*);

// construct a new NULL datavar type from a string
DFESP_API dfESPdatavar(dfESPdatatype *data_type*, const char **s*);

// construct a new NULL datavar type from a string
DFESP_API dfESPdatavar(char **dateFormat*, dfESPdatatype *data_type*, const char *s);

Required Arguments

data_type
can be one of the following values:

- **`ESP_INT32`**
- **`ESP_INT64`**
- **`ESP_DOUBLE (IEEE)`**
- **`ESP_UTF8STR`**
- **`ESP_DATETIME`** (second granularity)
- **`ESP_TIMESTAMP`** (microsecond granularity)
- **`ESP_MONEY`** (192-bit fixed decimal)

A **`dfESPdatavar`** of any of these types can be NULL. Two **`dfESPdatavars`** that are each NULL are not considered equal if the respective types do not match.

dateFormat
specifies the date format to use — if unspecified, the field value is interpreted as an integer number of seconds (**`ESP_DATETIME`**) or microseconds (**`ESP_TIMESTAMP`**) since epoch

s
specifies the string used to convert to the new data type

dv
specifies the existing data variable to replicate

b
specifies the binary data to convert

Example

Create an empty **`dfESPdatavar`** and then set the value as follows:

```
dfESPdatavar *dv = new dfESPdatavar(dfESPdatavar::ESP_INT32);
dv->setI32(13);
```

Get access to raw data in the **`dfESPdatavar`** using code like this:

```
void *p = dv->getRawdata(dfESPdatavar::ESP_INT32);
```

This returns a pointer to actual data, so in this **int32** example, you can follow with code like this:

```
int32_t x;
memcpy((void *)&x, p, sizeof(int32_t));
```

This copies the data out of the **dfESPdatavar**. You can also use the **getI32** member function (a get and set function exists for each data type) as follows:

```
int32_t x;
x = dv->getI32();
```

Many convenience functions are available and a complete list is available in the modeling API documentation included with the installation.

dfESPschema

represents a set of fields that together define event structures. Schemas include the specification of one or more key fields for the event structure that it defines. Each schema field has a name, a data type, and an indication as to whether it is part of the event key.

Syntax

DFESP_API dfESPschema(dfESPstring *id*);
DFESP_API dfESPschema(dfESPstring *id*, dfESPstring *schemaString*);

Required Arguments

id
user-supplied ID of the schema

schemaString
specify a serialized representation of the schema. Use the following form:
field1[*]:*type*,...,fieldn[*]:*type*

Details

A **dfESPschema** never represents field data, only the structure of the fields. When a **dfESPschema** object is created, it maintains the field names, fields types, and field key designation in the original order the fields (called the external order) and in the packing order (called the internal order) for the fields.

SAS Event Stream Processing does not put restrictions on the field names in a schema. Even so, you need to keep in mind that many times field names are used externally. For example, there could be connectors or adapters that read the schema and use field names to reference external columns in databases or in other data sources. It is therefore highly recommenced that you start field names with an alphanumeric character, and use no special characters within the name.

In external order, you specify the keys to be on any fields in the schema. In internal order, the key fields are shifted left to be packed at the start of the schema. For example, using a compact character representation where an "*" marks key fields, you specify this:

```
"ID1*:int32,symbol:string,ID2*:int64,price:double"
```

This represents a valid schema in external order. If you use this to create a **dfESPschema** object, then the object also maintains the following:

```
"ID1*:int32, ID2*:int64,symbol:string,price:double"
```

This is the same schema in internal order. It also maintains the permutation vectors required to transform the external form to the internal form and vice versa.

Creating a **dfESPschema** object is usually completed by passing the character representation of the schema to the constructor in external order, for example:

```
dfESPschema *s = new
     dfESPschema("mySchema","ID1*:int32,symbol:string,ID2*:
     int64,price:double");
```

A variety of methods are available to get the names, types, and key information of the fields in either external or internal order. There are also methods to serialize the schema back to the compact string form from its internal representation.

dfESPeventblock

An event block typically contains one or more events. Event blocks are published into source windows and maintained as a container with a unique ID throughout the processing of it within continuous queries. Events within event blocks usually are transformed as event blocks are processed by windows within a continuous query.

Syntax

// create an event block containing this single event
DFESP_API static dfESPeventblockPtr
*dfESPeventblock::newEventBlock(dfESPeventPtr ep);

// create an event block of this type for the event pointer list provided
DFESP_API static dfESPeventblockPtr
*dfESPeventblock::newEventBlock(dfESPptrList<dfESPeventPtr> *lst,
dfESPeventblocktype type);

// create an event block of this type for the event pointer vector provided
DFESP_API static dfESPeventblockPtr
*dfESPeventblock::newEventBlock(dfESPptrVect<dfESPeventPtr> *vec,
dfESPeventblocktype type);

// create a duplicate event block
DFESP_API static dfESPeventblockPtr
*dfESPeventblock::newEventBlock(dfESPeventblockPtr eb);

Required Arguments

ep
specifies an event pointer

type
specifies the event block type (**dfESPeventblock::ebtTRANS** | **ebtNORMAL)**

lst
specifies the event pointer list

vec
specifies the event pointer vector

eb
specifies the event block pointer

Details

Publishing clients can use this object to generate a **dfESPeventblock** object. An event block is maintained as it is passed between windows in an application, as well as to subscribing clients. The **dfESPeventblock** object can report the number of items that it contains and return a pointer to a contained **dfESPevent** when given an index.

A unique embedded transaction ID is generated for event blocks as they are absorbed into a continuous query. Event blocks can also be assigned a unique ID by the publisher. In addition to the event block ID, the publisher can set a host and port field in event blocks to establish where the event block is coming from. This meta information is used by the guaranteed delivery feature to ensure that event blocks make their way from a publisher.

Event blocks progress through the continuous queries and on to one or more guaranteed subscribers. The event block meta information is carried with the event block from the start of processing at a source window. The meta information progresses through all stages of computation in derived windows and on to any subscribing clients. You can use the publisher assigned ID, host, and port to tie an output **dfESPeventblock** back to an input **dfESPeventblock**.

Create new **dfESPeventblock** objects with either transactional (**dfESPeventblock::ebt_TRANS**) or normal (**dfESPeventblock::ebt_NORMAL**) event semantics. Transaction semantics imply that each **dfESPevent** contained in the block must be able to be applied to the index in a given window. Otherwise, none of the events are applied to the index.

For example, suppose an **dfESPeventblock** has 100 events and the first event is a delete event. Further suppose that the delete event fails to be applied because the underlying event to be deleted is not present. In that case, the remaining 99 events are ignored, logged, and written to a bad records file (optional). Normal semantics imply that each event in a **dfESPeventblock** is treated as an individual event. Therefore, the failure for one event to apply to an index causes only that event to not be incorporated into the window.

A **dfESPeventblock** with more than one event, but without transactional properties set, can be used to improve performance during the absorption of the event block into the appropriate source window. You use this to trade off a little bit of latency for a large gain in throughput. It is best to test the event block optimal size trade-off. For example, placing 256 events into an event block gives both great latency and throughput. This performance varies depending on the width of the events.

dfESPproject

specifies a container that typically holds one or more continuous queries. A project is backed by a thread pool of user-defined size and an optional port.

Syntax

```
// create project with given id, tagged token off, and no depot location for caching store
DFESP_API dfESPproject *dfESPengine::newProject(dfESPstring id);

// create project with id, tagged token on or off,
// and depot location (which could be an empty string for none)
DFESP_API dfESPproject
*dfESPengine::newProject(dfESPstring id, bool useTaggedToken, dfESPstring
depotLocation);
```

Required Arguments

id
specifies the project ID

useTaggedToken
indicates whether to use project tagged token

depotLocation
specifies where to place the project caching store or stores when a window uses a caching index (for example, **`piHLEVELDB`**). This can be an empty string for no caching store.

Details

The levels of determinism supported by a project are as follows:

- full concurrency (default) - data received by a window is processed as soon as it is received and forwarded on to any dependent window. This is the highest performing mode of computation. In this mode, a project can use any number of threads as specified by the **`setNumberOfThreads(max thread)`** method.
- tagged token - implements single-transaction in, single-transaction out semantics at each node of the model. In this mode, a window imposes a diamond patter, splitting the output and then rejoining the split paths together. It merges outputs (per unique transaction) from each path into a single transaction. A single transaction in the top of the diamond produces a single output at the bottom.

The **`newProject()`** method for the **`dfESPengine`** class takes a final parameter (**`true`** | **`false`**) that indicates whether tagged token data flow should be enabled. If you do not specify this optional parameter, the value defaults to false.

- Thus, to specify full concurrency:

```
dfESPproject *project = engine->newProject("MyProject”);
```

or

```
dfESPproject *project = engine->newProject("MyProject”, false);
```

- And to specify tagged token:

```
dfESPproject *project = engine->newProject("MyProject”, true);
```

For easier debugging and full consistency in output for testing, run with tagged token **`true`**. Set the number of threads in a project to 1. This is the slowest way to run a project. Nonetheless, as long as you are using time-based retention policies, you can be assured that the output is consistent from run to run.

Example

The following code fragment shows how to create a project, add a memory store, set the thread pool size, and add continuous queries. It is run with a level of determinism of full consistency.

```
// Create the project containers.
dfESPproject *project = engine->newProject("MyProject”);

project->setNumThreads(3);  //set the thread pool size for project.

// After you have started the projects using the startProjects()
//     method shown  in the dfESPengine section above, then you
```

```
//     can publish or use dfESPproject::injectData() to inject
//     event blocks into the source windows. You can also use
//     the dfESPproject::quiesce() method to block until all
//     of the data in the continuous queries of the project are
//     quiesced. You might also want to do this before stopping
//     the projects.

project->quiesce();
project->stopProject();
```

dfESPcontquery

specifies a container that holds one or more directed graphs of windows. Many projects have a single continuous query container. You can use continuous queries to implement functional modularity for large projects. You can use the project connector to publish event streams from a window in one continuous query to a source window in another continuous query, in the same or in another project.

Syntax

DFESP_API dfESPcontquery *dfESPproject::newContquery(dfESPstring id);

Required Argument

id

specifies the continuous query ID

Example

Suppose that there are two windows, **`swA`** and **`swB`**, that are joined to form window **`jwC`**. Window **`jwC`** is aggregated into window **`awD`**. Build the continuous query as follows, using the **`addEdge`** function:

```
dfESPcontquery *cq;
cq = project->newContquery("continuous query #1");

cq->addEdge(swA, jwC); // swA --> jwC
cq->addEdge(swB, jwC); // swB --> jwC
cq->addEdge(jwC, awD); // jwC --> awD
```

This fully specifies the continuous query with window connectivity, which is a directed graph.

dfESPwindow_source

specifies a window that is used to ingest event streams of a defined schema into a continuous query. Source windows can have retention policies. Event streams can be published only into source windows.

Syntax

DFESP_API dfESPwindow_source
*dfESPcontquery::newWindow_source(dfESPstring *id*, pindex_t *index*, dfESPstring *schema*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows*.

schema
user-supplied string that specifies the structure of the fields of event data.

Details

The source window is the only window type to which event streams can be published. All other window types are called derived windows, which transform or analyze event streams coming into them from other windows.

Example

Here is an example of how to specify a source window:

```
dfESPwindow_source *sw;
dfESPstring sch = dfESPstring("ID*:int32,symbol:string,price:double");
sw = cq->newWindow_source("mySourceWindow, dfESPindextypes::pi_HASH, sch);
```

You can set event state retention for source windows and copy windows 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 automatically by the engine when they exceed the window's retention policy.

Set the retention type on a window with the **`setRetentionParms()`** call. You can set type by count or time, and as either jumping or sliding.

Under the following circumstances, a source window can auto-generate the key value:

- the source window is Insert only
- there is only one key for the window
- the key type is INT64 or string

When these conditions are met and the **`setAutoGenerateKey()`** call is made, you do not have to supply the key value for the incoming event. The source window overwrites the value with an automatically generated key value. For INT64 keys, the value is an incremental count (0, 1, 2, ...). For STRING keys, the value is a Globally Unique Identifier (GUID).

dfESPwindow_aggregate

specifies a window that aggregates events from its incoming event stream. Aggregation is based on the key fields specified for the aggregate window schema. The result is a collection of groups with aggregated fields.

Syntax

DFESP_API dfESPwindow_aggregate

*dfESPcontquery::newWindow_aggregate(dfESPstring *id*, pindex_t *index*, dfESPstring *schema*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

schema
specifies an aggregate schema. The specification is the same as for any other window schema, except that key field(s) are the group-by mechanism.

See Also

"Creating Aggregate Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_compute

specifies a window that enables users to define projections or transformations on input events fields in order to produce new compute window events. There is a one-to-one cardinality between input events and generated events for this window type.

Syntax

DFESP_API dfESPwindow_compute
*dfESPcontquery::newWindow_compute(dfESPstring *id*, pindex_t *index*, dfESPstring *schema*);

Required Arguments

id
user-supplied identifier of the compute window

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

schema
user-supplied name of the schema as specified by **`dfESPstring`**.

Details

Usually, the keys of a compute window are obtained from the keys of its input window. However, key values can be changed by designating the new fields as key fields in the **`dfESPcompute_window`** schema. When you change the key value in the compute window, the new key must also form a primary key for the window. If it does not, you might encounter errors because of unsuccessful Insert, Update, and Delete operations.

Examples

Example 1

Here is an example of a specification of a compute window:

```
dfESPwindow_source *cw;
cw = cq->newWindow_compute("myComputeWindow",
     dfESPindextypes::pi_HASH, sch);
```

As with the source window, you use **dfESPstring** to specify a schema. For example

```
dfESPstring sch = dfESPstring("ID*:int32,symbol:string,price:double");
```

A compute window needs a field calculation method registered for each non-key field so that it computes the field value based on incoming event field values. These field calculation methods can be specified as either of the following:

- a collection of function pointers to C or C++ functions that return **dfESPdatavar** values and are passed an event as input
- expressions that use the field names of the input event to compute the values for the derived event fields

Example 2

The following example creates a compute window using a collection of function pointers.

Assume the following schema for input events:

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

The compute window passes through the input symbol and price fields. Then it adds a computed field (called cost) to the end of the event, which multiplies the price with the quantity.

A scalar function provides the input event and computes *price* * *quantity*. Functions that take events as input and returns a scalar value as a **dfESPdatavar** use a prototype of type **dfESPscalar_func** that is defined in the header file **dfESPfuncptr.h**.

Here is the scalar function:

```
dfESPdatavar *priceBYquant(dfESPschema*is, dfESPevent *nep,
      dfESPevent *oep, dfESPscontext *ctx) {
   //
   // If you are getting an update, then nep is the updated
   //      record, and oep is the old record.
   //
   // Create a null return value that is of type double.
   //
   dfESPdatavar *ret = new dfESPdatavar(dfESPdatavar::ESP_DOUBLE);
   // If you were called because a delete is being issued, you do not
   //      compute anything new.
   //
   if (nep->getOpcode() == dfESPeventcodes::eo_DELETE)
      return ret;
   void *qPtr = nep->getPtrByIntIndex(2); // internal index of
        quant is 2
   void *pPtr = nep->getPtrByIntIndex(3); // internal index of
        price is 3
   if ((qPtr != NULL) && (pPtr != NULL)) {
      double price;
```

```
        memcpy((void *) &price, pPtr, sizeof(double));
        int32_t quant;
        memcpy((void *) &quant, qPtr, sizeof(int32_t));
        ret->setDouble(quant*price);
    }
    return ret;
}
```

Note the **dfESPscontext** parameter. This parameter contains the input window pointer, the output schema pointer, and the ID of the field in the output schema computed by the function. Parameter values are filled in by the engine and passed to all compute functions. Go to **$DFESP_HOME/examples/cxx/compute_context** for another example that shows how to use this parameter.

The following code defines the compute window and registers the non-key scalar functions:

```
dfESPstring sch =
       dfESPstring("ID*:int32,symbol:string, price:double,cost:double");

dfESPwindow_compute *cw;
cw = cq->newWindow_compute("myComputeWindow",
                           dfESPindextypes::pi_HASH, sch);

// Register as many function pointers as there are non-key
//      fields in the output schema. A null for non-key
//      field j means copy non-key field j from the input
//      event to non-key field j of the output event.
//
cw->addNonKeyFieldCalc((dfESPscalar_func)NULL); // pass
     through the symbol
cw->addNonKeyFieldCalc((dfESPscalar_func)NULL); // pass
     through the price value
cw->addNonKeyFieldCalc(priceBYquant); // compute
     cost = price * quantity
```

This leaves a fully formed compute window that uses field expression calculation functions.

Example 3

The following example creates a compute window using field calculation expressions rather than a function. It uses the same input schema and compute window schema with the following exceptions:

1. You do not need to write field expression calculation functions.
2. You need to call **addNonKeyFieldCalc()** using expressions.

Note: Defining the field calculation expressions is typically easier. Field expressions can perform slower than calculation functions.

```
dfESPstring sch =
       dfESPstring("ID*:int32,symbol:string,price:double,cost:double");

dfESPwindow_compute *cw;
cw = cq->newWindow_compute("myComputeWindow",
                           dfESPindextypes::pi_HASH, sch);

// Register as many field expressions as there are non-key
```

```
//      fields in the output schema.
cw->addNonKeyFieldCalc("symbol"); // pass through the symbol
     value
cw->addNonKeyFieldCalc("price"); // pass through the price
     value
cw->addNonKeyFieldCalc("price*quantity"); // compute cost
     = price * quantity
```

Note: The field calculation expressions can contain references to field names from the input event schema. They do not contain references to fields in the compute window schema. Thus, you can use similarly named fields across these schemas (for example, symbol and price).

Note: Currently, you cannot specify both field calculation expressions and field calculation functions within a given window.

For more information, see the *DataFlux Expression Language: Reference Guide*.

dfESPwindow_copy

specifies a window that maintains a copy of the events from its parent window. Copy windows inherit their parent's schema. Copy windows are typically used to establish new retention policies.

Syntax

DFESP_API dfESPwindow_copy
*dfESPcontquery::newWindow_copy(dfESPstring *id*, pindex_t *index*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

Details

Set the retention type on a window with the **`setRetentionParms()`** call. You can set type by count or time, and as either jumping or sliding. You can define retention policies only in source and copy windows.

Example

Here is an example of how to specify a copy window:

```
dfESPwindow_copy *cw;
cw = cq->newWindow_copy("myCopyWindow",
     dfESPindextypes::pi_HASH);
```

You can set event state retention for copy windows 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.

Set the retention type on a window with the **`setRetentionParms()`** call. You can set type by count or time, and as either jumping or sliding.

See Also

"Creating Copy Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_counter

specifies a window that determines event volumes and throughput rate over a defined recurring interval.

Syntax

DFESP_API dfESPwindow_counter
*dfESPcontquery::newWindow_counter(dfESPstring *id*, pindex_t *index*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

Details

The schema for counter windows is fixed as follows:

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

The key field **`input`** is the input window ID, so that there is always only one event per parent input window.

Example

```
dfESPproject*project = engine->newProject("project");
dfESPcontquery*contquery = project->newContquery("contquery");
dfESPwindow_source*source =
 contquery->newWindow_source("source",dfESPindextypes::pi_RBTREE,
                             dfESPstring("ID*:int32,symbol:string,valstr:string"));
dfESPschema *schema = source->getSchema();
dfESPwindow_counter *counter = contquery->newWindow_counter("counterWindow",
                                          dfESPindextypes::pi_RBTREE);
counter->setCountInterval("2 seconds");
counter->setClearInterval("10 seconds");
contquery->addEdge(source,0,counter);
```

See Also

"Creating Counter Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_filter

specifies a window that filters the incoming event stream based on a filter expression or a user-defined function

Syntax

DFESP_API dfESPwindow_filter
*dfESPcontquery::newWindow_filter(dfESPstring *id*, pindex_t *index*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

Details

The filter function or expression is set by the **`dfESPwindow_filter::setFilter()`** method. This function or expression is called each time that a new event block arrives in the filter window. The function or expression uses the fields of the events that arrive to determine the Boolean result. If the function or expression evaluates to true, then the event passes through the filter. Otherwise, the event does not pass into the filter window.

There are two ways to specify the Boolean filter associated with a filter window:

- through a C function that returns a **`dfESPdatavar`** of type int32 (return value != 0 ==> true; == 0 ==> false)
- by specifying an expression as a character string so that when it is evaluated it returns true or false

Examples

Example 1

The following example writes and registers a filter user-defined function:

```
// When quantity is >= 1000, let the event pass
//
//
dfESPdatavarPtr booleanScalarFunction(dfESPschema *is,
dfESPeventPtr ep, dfESPeventPtr oep) {

     // Get the input argument out of the record.
     dfESPdatavar dv(dfESPdatavar::ESP_INT32);
     // Declare a dfESPdatavar that is an int32.
     ep->copyByIntID(2, dv); // extract field #2 into the datavar

     // Create a new dfESP datavar of int32 type to hold the
```

```
//      0 or 1 that this Boolean function returns.
//
dfESPdatavarPtr prv = new dfESPdatavar(dfESPdatavar::ESP_INT32);

// If field is null, filter always fails.
//
if (dv.isNull()) {
     prv->setI32(0); // the return value to 0
} else {
     // Get the int32 value from the datavar and compare to 1000
     if (dv.getI32() < 1000) {
          prv->setI32(0);     // set return value to 0
     } else {
          prv->setI32(1);     // set return value to 1
     }
}
return prv; // return it.
```

Place the following code inside **`main()`**:

```
dfESPwindow_filter *fw_01;
     fw_01 = cq->newWindow_filter("filterWindow_01",
                                  dfESPindextypes::pi_RBTREE);
     fw_01->setFilter(booleanScalarFunction);
       // Register the filter UDF.
```

The **`setFilter`** function calls the filter function named booleanScalarFunction that you had previously registered.

Example 2

The following code example uses filter expressions.

```
dfESPwindow_filter *fw_01;
fw_01 = cq->newWindow_filter("filterWindow_01",
                             dfESPindextypes::pi_RBTREE);
fw_01->setFilter("quant>=1000");
// Register the filter expression.
```

For more information about user-supplied filter expressions, see the *DataFlux Expression Language: Reference Guide*.

See Also

"Creating Filter Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_functional

specifies a window that enables a user to specify transformations and manipulations of incoming events. Transformations and manipulations are performed through specific functions, and can result in each input event generating zero or more functional window events. Incoming event string fields could contain hierarchical data such as JSON or XML.

Syntax

DFESP_API dfESPwindow_functional
*dfESPcontquery::newWindow_functional(dfESPstring *id*, pindex_t *index*, dfESPstring *schema*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

schema
user-defined functional window schema

Details

For more information about the functions that you can use to transform or manipulate incoming events, see Chapter 2, "Functional Window and Notification Window Support Functions," .

For technical details, see "Creating Functional Windows" in *SAS Event Stream Processing: Creating and Using Windows* .

dfESPwindow_join

specifies a window that joins two incoming event streams based on the specified join type and condition

Syntax

DFESP_API dfESPwindow_join
*dfESPcontquery::newWindow_join(dfESPstring *id*, joinsub_t *jt*, pindex_t *index*, bool useSecondary=true | false, bool noregenerates=true | false);

Required Arguments

id
specifies the window ID

jt
type of join (**`dfESPwindow_join::dfESPjointypes`**) to be applied

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows*.

Optional Arguments

useSecondary
a Boolean value that determines whether the join window should auto-generate a secondary index. The default value is false.

noregenerates
a Boolean value that, when true, implies changes to the dimensional window and does not cause a rescan of the fact window to produce a block of lookup updates. The default value is false

Details

The key values of the join-window schema are calculated depending on the join type, join condition, and input window schema keys. Join windows support equijoins 1-1 (left, right, or full outer or inner), 1-M and M-1 (left, right, outer or inner), and M-M (inner). Specify the join condition using the **`setJoinConditions()`** member function of the join window class.

See Also

"Creating Join Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_notification

species a window that enables the event stream processing engine to send notifications through email, text, or multimedia messages. It enables the creation of a number of delivery channels through which to send notifications. Notification windows do not generate events, and are workflow in nature. They have a schema that is used to support the functions used in the notification window.

Syntax

DFESP_API dfESPwindow_notification
*dfESPcontquery::newWindow_notification(dfESPstring *id*, dfESPstring *schema*);

Required Arguments

id
specifies the window ID

schema
optional window schema that is strictly used by notification window functions. The value can be NULL.

Details

For more information about the functions that you can use, see Chapter 2, "Functional Window and Notification Window Support Functions," .

For more information, see "Creating Notification Windows" in *SAS Event Stream Processing: Creating and Using Windows* .

Example

The following code sample sets up an email notification to a broker.

```
dfESPwindow_notification *notification =
   contquery->newWindow_notification("notify",NULL);
   notification->setSmtpConnection("mailhost.fyi.orion.com");
```

```
dfESPemail    *email = notification->addEmail();
email->setThrottleInterval("5 minutes");
email->setSender("john.doe@orion.com");
email->setRecipients("$email");
email->setSubject("Investment Opportunity");
email->setFrom("ESP");
email->setTo("Wealthy Trader");
email->addText("You traded $quant shares of $symbol at $$price.");
email->setTestMode(true);

contquery->addEdge(joinBrokerData,0,notification);
```

dfESPwindow_pattern

specifies a window that enables the creation of one or more pattern definitions. Patterns are defined to detect correlations of multiple events across a set of events-of-interest conditions and temporal conditions.

Syntax

DFESP_API dfESPwindow_pattern
*dfESPcontquery::newWindow_pattern(dfESPstring *id*, pindex_t *index*, dfESPstring *schema*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

schema
schema of the pattern window

See Also

"Creating Pattern Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_procedural

specifies a window that enables users to register event stream input handlers for the window. The input handlers process incoming event streams using C++, DS2, or callouts to Base SAS through DATA step statements. Procedural windows can have one or more input event streams, each requiring its own input handler.

Syntax

DFESP_API dfESPwindow_procedural

*dfESPcontquery::newWindow_procedural(dfESPstring *id*, pindex_t *index*, dfESPstring *schema*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

schema
schema of the procedural window defined by **`dfESPstring`**

See Also

"Creating Procedural Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_textCategory

specifies a window that enables you to use a SAS Text Analytics MCO file to get the categories of a document in each event's specified string field. This window type is insert only. For each input event, the window generates zero or more category events.

Syntax

DFESP_API dfESPwindow_textCategory
*dfESPcontquery::newWindow_textCategory(dfESPstring *id*, pindex_t *type*,
dfESPstring *mcoFile*,
dfESPstring *stringField*, bool nullEvents=true | false);

Required Arguments

id
specifies the window ID

type
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

mcoFile
path to the MCO file

stringField
user-supplied name for the input string field to analyze

nullEvents
Boolean flag to determine whether to generate a NULL event when no categories are found for an incoming event. The default value is FALSE.

Details

Because this window is insert-only, the following is recommended:

- you follow the window with a copy window so that events can be retained as needed
- you use an empty index for the window

You need a SAS Contextual Analysis license so that you can provide the required *mcoFile* .

Text category windows generate their own schema, which is as follows:

```
key_fields_of_input_window, CategoryNumber*:int32, Category:string, score:double.
```

The key fields for the window are *key_fields_of_input_window* and CategoryNumber.

See Also

"Creating Text Category Windows" in *SAS Event Stream Processing: Creating and Using Windows*

dfESPwindow_textContext

specifies a window that uses SAS Text Analytics LITI files to extract text and classify terms using a specified input event string field. This window type is insert only. For each input event, zero or more events are created for this window, depending on how many classified terms are discovered by the text engine.

Syntax

DFESP_API dfESPwindow_textContext
*dfESPcontquery::newWindow_textContext(dfESPstring *id*, pindex_t *index*,
dfESPstring *initFiles*,
dfESPstring *fieldName*, bool nullEvents=true | false);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary index are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

initFiles
comma separated list of Liti file paths or a single path to an aStore.

fieldName
user-supplied name for the string field in the input event to analyze

nullEvents
Boolean flag to determine whether to generate a NULL event when no categories are found for an incoming event. The default value is FALSE.

Details

You need a SAS Contextual Analysis license so that you can provide the required *initFiles*.

The schema of events output by the **`textContext`** window is as follows:

```
"input event key fields*, termID*:int64, term:string, termLen:int32,
```

```
 tag:string, tagLen:int32, locStart:int32, locEnd:int32, wordStart:int32,
wordEnd:int32"
```

Example

The following example uses an empty index for the **`textContext`** window, which is insert only. That way, the window does not grow endlessly. The **`textContext`** window is followed by a copy window that uses retention to control the growth of the classified terms.

```
// Build the source window. We specify the window name, the schema
// for events, the depot used to generate the index and handle
// event storage, and the type of primary index, in this case a
// hash tree
//
dfESPwindow_source *sw_01;
  sw_01 = cq_01->newWindow_source("sourceWindow_01",
                                  dfESPindextypes::pi_HASH,
                                  dfESPstring("ID*:int32,tstamp:date,msg:string"));

// Build the textContext window. Specify the window name, the depot
// used for retained events, the type of primary index, the Liti
// files specification string, and the input string field to analyze.
//
// Note that the index for this window is an empty index.  This means
// that events created in this window will not be retained in this
// window.  This is because textContext windows are insert only,
// hence there is no way of deleting these events using retention
// so without using an empty index this window would grow indefinitely.
//
// To run this example, you need to have licensed SAS
// Contextual Analysis, whose install contains these Liti language files.
// You need to change the litFiles string below to point to your
// installation of the Liti files. Otherwise the text analytics engine
// will not be initialized and classified terms will not be found.
//
dfESPwindow_textContext *tcw_01;

dfESPstring initFiles = "/wire/develop/TableServer/src/common/dev/
                         mva-v940m1/tktg/misc/en-ne.li,
                         /wire/develop/TableServer/src/common/dev/
                         mva-v940m1/tktg/misc/ro-ne.li";

// Place a copy window after the textContext window so that
// it can be used to hold the textContext events with an established
// retention policy. This is a design pattern for insert-only windows.
tcw_01 = cq_01->newWindow_textContext("textContextWindow_01",
                                      dfESPindextypes::pi_EMPTY,
                                      initFiles, "msg");

// Create the copy window.
dfESPwindow_copy *cw_01;
cw_01 = cq_01->newWindow_copy("copyWindow_01",
                              dfESPindextypes::pi_RBTREE);

// Now set the window's retention policy to a sliding window of 5 mins.
```

```
// This example only has 3 events being injected so the retention
//    policy will not take effect, but if we published enough data
//    into this model then it would start retaining older events out
//    using retention deletes once they aged past 5 mins.
cw_01->setRetentionParms(dfESPindextypes::ret_BYTIME_SLIDING, 300);
```

Suppose you supply the following strings to the **textContext** window:

```
"i,n,1,2010-09-07 16:09:01,I love my Nissan pickup truck"
"i,n,2,2010-09-07 16:09:21,Jerry went to dinner with Renee for last Sunday"
"i,n,3,2010-09-07 16:09:43,Jennifer recently got back from Japan where
 she did game design project work at a university there"
```

Here are the results.

```
event[0]: <I,N: 1,0,Nissan pickup,13,VEHICLE,7,10,22,3,5>
event[1]: <I,N: 2,0,Sunday,6,DATE,4,41,46,8,9>
event[2]: <I,N: 2,1,Renee,5,PROP_MISC,9,26,30,5,6>
event[3]: <I,N: 3,0,Japan,5,LOCATION,8,32,36,5,6>
event[4]: <I,N: 3,1,game design project work,24,NOUN_GROUP,10,52,75,9,13>
event[5]: <I,N: 3,2,Jennifer,8,PROP_MISC,9,0,7,0,1>
```

dfESPwindow_textSentiment

specifies a window that uses a SAS Text Analytics SAM file to get the sentiment of a document that is in the specified incoming event string field. This window type is insert only. Each input event creates a new text sentiment event.

Syntax

DFESP_API dfESPwindow_textSentiment
*dfESPcontquery::newWindow_textSentiment(dfESPstring *id*, pindex_t *index*,
dfESPstring *samFile*,
dfESPstring *inputFieldName*);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

samFile
path to the SAM file

inputFieldName
user-supplied name of the string field in the input event to analyze

Details

Because this window is insert-only, the following is recommended:

- you follow the window with a copy window so that events can be retained as needed
- you use an empty index for the window

You need a SAS Sentiment Analysis license so that you can provide the required *samFile* .

The window generates its own schema, which is as follows: ***key_fields_of_input_window*, sentiment:string, probability:double**.

The sentiment value is "**positive**", "**neutral**", or "**negative**", and the probability is a value between 0 and 1.

dfESPwindow_union

specifies a window that unites two or more event streams using a strict or a loose policy

Syntax

DFESP_API dfESPwindow_union
*dfESPcontquery::newWindow_union(dfESPstring *id*, pindex_t *index*, bool strict=true | false);

Required Arguments

id
specifies the window ID

index
primary index. Six types of primary indexes are supported. For more information, see "Understanding Primary and Specialized Indexes" in *SAS Event Stream Processing: Creating and Using Windows* .

Optional Argument

strict
the strict flag — true for strict union and false for loose unions. The default value is false.

Details

Union windows support both a strict union and a loose union of events from two or more parent windows. The loose variant (strict = false) turns Updates into Upserts and Deletes into Safe Deletes. A Safe Delete fails quietly when the event to be deleted does not exist.

Example

Here is an example of how to create a union window:

```
dfESPwindow_union *uw;
uw = cq->newWindow_union("myUnionWindow",
     dfESPindextypes::pi_HASH, true);
```

Chapter 2

Functional Window and Notification Window Support Functions

Dictionary

ABS

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

Syntax

abs(*argument*)

Required Argument

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.

Example

```
abs(-55)=55
abs(44)=44
```

AND

When both arguments are true, returns true. Otherwise, returns false.

Syntax

and(*argument1*, *argument2*)

Required Argument

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 would 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.

Example

```
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.

Syntax

base64Decode(*string*)

Required Argument

string
specifies a base64–encoded string. There is no length limit to this string.

Example

```
base64Decode('dGhpcyBpcyBhIHRlc3Q=')=this is a test
```

BASE64ENCODE

Encodes the supplied string into a base64–encoded string.

Syntax

base64Encode(*string*)

Required Argument

string
specifies a text string. There is no length limit to this string.

Example

```
base64Encode('this is a test')=dGhpcyBpcyBhIHRlc3Q=
```

BETWEEN

If the first argument is greater than the second and less than the third, returns true. Otherwise, returns false.

Syntax

between(*argument1*, *argument2*, *argument3*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
between(20,17,30)=1
between(20,17,15)=0
```

BOOLEAN

If the supplied argument is a string, returns true when the string has length greater than 0. If the supplied argument is numeric, returns true when value is not equal to 0. If the supplied argument is a Boolean expression, returns true when the value is true. Otherwise, returns false.

Syntax

boolean(*argument*)

Required Argument

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.

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.

Example

```
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.

Syntax

ceiling(*argument*)

Required Argument

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.

Example

```
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.

Syntax

compare(*argument1*, *argument2*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

The type of the first argument determines whether equality is determined by a string or numeric comparison.

Example

```
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.

Syntax

concat(*argument1*, *argument2*,…<*argumentN*>)

Required Argument

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 would specify this: **#myXML**.
- a function.

A minimum of two arguments is required.

Example

```
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.

Syntax

concatDelim(‘*delimiter*’,*argument1*, *argument2*, …<*argumentN*>)

Required Arguments

‘*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 would specify this: **#myXML**.
- a function.

Example

```
concatDelim('.','www','sas','com') = www.sas.com
```

CONTAINS

If the string value of the first argument contains the string value of the second, then it returns true. Otherwise, it returns false.

Syntax

contains(*argument1*, *argument2*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
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.

Syntax

decrement(*argument*)

Required Argument

argument
specifies an integer.

Example

```
decrement(10)=9
```

DIFF

Returns the value of the first argument minus the second.

Syntax

diff(*argument1*, *argument2*)

Required Argument

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 would specify this: **#myXML**.
- a function.

Example

```
diff(sum(8,7),22)=-7.0
```

EQUALS

Returns true if the first argument is equal to the second. Otherwise, it returns false.

Syntax

equals(*argument1*, *argument2*)

Required Argument

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 would specify this: **#myXML**.
- a function.

The type of the comparison depends on the type of the first argument.

Example

```
equals('sas.com',string('sas','.com'))=1
equals('sas.com','google.com')=0
equals(10,sum(5,5))=1
```

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.

Syntax

eventNumber()

Example

```
string($id,'-',eventNumber())=eventid-0
string($id,'-',eventNumber())=eventid-1
string($id,'-',eventNumber())=eventid-2
```

FALSE

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

Syntax

false(*argument*)

Required Argument

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.

Example

```
false(0)=1
false(equals(10,10))=0
```

FLOOR

Returns the integer value below the numeric value of the supplied argument.

Syntax

floor(*argument*)

Required Argument

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.

Example

```
floor(product(3.5,7))=24
```

GT

Returns true if the first argument is greater than the second. Otherwise, it returns false.

Syntax

gt(*argument*, *argument2*)

Required Argument

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 would specify this: **#myXML**.
- a function.

The type of the comparison depends on the type of the first argument.

Example

```
gt(sum(10,4),13)=true
gt('internet explorer','internet explorer')=false
gt('internet explorer','netscape')=false
```

GTE

Returns true if the first argument is greater than or equal to the second. Otherwise, it returns false.

Syntax

gt(*argument*, *argument2*)

Required Argument

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 would specify this: **#myXML**.
- a function.

The type of the comparison depends on the type of the first argument.

Example

```
gte(sum(10,4),13)=true
gte('internet explorer','internet explorer')=true
gte('netscape','internet explorer')=true
```

GUID

Returns a globally unique identifier.

Syntax

guid()

Example

```
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.

Syntax

increment(*argument*)

Required Argument

argument
specifies an integer value or a function that returns an integer value.

Example

```
increment(10)=11
```

IF

If the Boolean value of the first argument is true, returns the second argument. Otherwise, it returns the third argument if specified.

Syntax

if(*argument1* , *argument2*, <*argument3*>)

Required Arguments

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 would specify this: **`#myXML`**.
- a function.

Optional Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
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.

Syntax

ifNext(*argument* , *argument2*, …<*argumentN*>, <*argumentN+1*>)

Required Arguments

argument
specifies a Boolean expression.

argument2
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

Example

```
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
```

INDEX

Returns the value of argumentN, where N is the numeric value of specified index.

Syntax

index(*index* , *argument0*, …<*argumentN*>)

Required Arguments

index
specifies an integer or a function that returns an integer.

argument0*, *argument1*, …*argumentN
specifies one of the following:

- a literal value, either string or numeric. Enclose string values in single or double quotation marks.
- a value to be resolved from an event field. Precede field names with the $ character.
- a value that refers to a resource. For example, when you use the **`xpath`** function to refer to an XML object named myXML, you would specify this: **`#myXML`**.
- a function.

The minimum number of arguments is 2.

Example

```
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 if the value is not found.

Syntax

indexOf(*argument1* , *argument2*)

Required Argument

argument1*, *argument2
specifies a string.

Example

```
indexOf('SAS Event Stream Processing','Stream')=10
indexOf('SAS Event Stream Processing','Google')=-1
```

INPUT

Returns the name of the event stream processing input window.

Syntax

input()

Example

```
input()=sourceWindow
```

INTEGER

Returns the integer value of the argument.

Syntax

integer(*argument*)

Required Argument

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.

Example

```
integer('88.45')=88
integer(111.23)=111
```

ISNULL

Returns true if the argument is not set, otherwise it returns false.

Syntax

isNull(*argument*)

Required Argument

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.

Example

```
isNull($unresolved)=1
isNull('my string')=0
```

ISSET

Returns true if the argument is set, otherwise it returns false.

Syntax

isSet(*argument*)

Required Argument

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.

Example

```
isSet($unresolved)=0
isSet('my string')=1
```

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.

Syntax

interval(*argument* , *argument2*, *argument3*, …<*argumentN*>)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
interval(85,60,'F',70,'D',80,'C',90,'B','A')=B
interval(90,60,'F',70,'D',80,'C',90,'B','A')=A
```

JSON

Parses the JSON object specified in the first argument and returns a value as a function of the second argument.

Syntax

json(*argument* , *argument2*)

Required Arguments

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.

Example

```
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 if the value is not found.

Syntax

lastIndexOf(*argument* , *argument2*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
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.

Syntax

listItem(*argument* , *delimiter*, *index*)

listItem(*reference* , *index*)

Required Arguments

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.

Example

```
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.

Syntax

listSize(*argument* , *delimiter*)

listSize(*reference*)

Required Arguments

argument
specifies a delimited string.

delimiter
specifies a character used as a delimiter.

reference
specifies a reference to a function context.

Example

```
listSize('one,two,three,four',',')=4
listSize(#myList)=4
```

LONG

Returns the long value of the argument.

Syntax

long(*argument*)

Required Argument

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.

Example

```
long('88.45')=88
long(111.23)=111
```

LT

Returns true if the first argument is less than the second. Otherwise, returns false.

Syntax

lt(*argument* , *argument2*)

Required Arguments

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.

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 would specify this: **#myXML**.
- a function.

Example

```
lt(sum(10,4),13)=false
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

LTE

Returns true if the first argument is less than or equal to the second. Otherwise, returns false.

Syntax

lt(*argument* , *argument2*)

Required Argument

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 would specify this: **#myXML**.
- a function.

Example

```
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.

Syntax

mapValues(*argument* , *outerdelimiter*, *innerdelimiter*, *delimiter*, *name*)

mapValues(*#reference*, *name*)

Required Arguments

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.

Example

```
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.

Syntax

mapValues(*argument* , *outerdelimiter*, *innerdelimiter*, *delimiter*, *name1*,…<*nameN*>)

mapValues(*#reference*, *name1*, …<*nameN*>)

Required Arguments

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.

Example

```
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.

Syntax

max(*argument* , *argument2*, *…<argumentN>*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

The minimum number of arguments is 1.

Example

```
max(33,44.2,sum(1,3,2,12),-33.21)=44.2
```

MEAN

Returns the mean value of all specified arguments.

Syntax

mean(*argument* , *argument2*, *…<argumentN>*)

Required Argument

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 would specify this: **#myXML**.
- a function.

Example

```
mean(33,44.2,sum(1,3,2,12),-33.21)=15.4975
```

MIN

Returns the smallest numeric value of all specified arguments.

Syntax

min(*argument* , *argument2*, …<*argumentN*>)

Required Argument

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 would specify this: **#myXML**.
- a function.

The minimum number of arguments is 1.

Example

```
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.

Syntax

mod(*argument* , *argument2*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
mod(10,3)=1.0
```

NEG

Returns the negative numeric value of the specified argument.

Syntax

neg(*argument*)

Required Argument

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.

Example

```
neg(55)=-55
```

NORMALIZESPACE

Returns a string that is created by replacing any extra white space in the specified argument with a single space.

Syntax

normalizeSpace(*argument*)

Required Argument

argument
specifies a string.

Example

```
normalizeSpace('Sentence   with   many  spaces')=Sentence with many spaces
```

NEQUALS

Returns true if the first argument is not equal to the second. Otherwise, returns false.

Syntax

nequals(*argument* , *argument2*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
nequals('sas.com',string('sas','.com'))=0
nequals('sas.com','google.com')=1
nequals(10,sum(5,5))=0
```

NOT

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

Syntax

not(*argument*)

Required Argument

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.

Example

```
not(0)=1
not(equals(10,10))=0
```

NUMBER

Returns the numeric value of the argument.

Syntax

number(*argument*)

Required Argument

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.

Example

```
number('88.45')=88.45
number(111.23)=111.23
number(gt(2,1))=1
```

OR

Returns true if any of the supplied arguments are true. Otherwise, returns false.

Syntax

or(*argument* , *argument2*, …<*argumentN*>)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

The minimum number of arguments is 1.

Example

```
or(equals('a','b'),nequals('a','b'))=1
or(equals('a','b'),nequals('a','a'))=0
```

OUTPUT

Returns the name of the event stream processing output window.

Syntax

output()

Example

```
input()=myFunctionalWindow
```

OUTSTR

If the argument contains a string or one of a group of strings, returns the associated value. If there are no matches, returns a specified default value.

Syntax

outstr(*argument* , *string1*, *value_associated_with_string1*, …<*stringN*>, <*value_associated_with_stringN*>, *default*)

Required Arguments

argument
specifies a string.

string1…stringN
specifies a string or group of strings.

value_associated_with_string1…value_associated_with_stringN
specifies a string or group of strings.

default
specifies a string or group of strings.

Example

```
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.

Syntax

precision(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies a numeric value.

Example

```
precision(123.44567,2)=123.45
```

PRODUCT

Returns the product of the supplied arguments.

Syntax

product(*argument* , *argument2…<argumentN>*)

Required Argument

argument*, *argument2*, … *argumentN
specifies a numeric value or a function that returns a numeric value.

Example

```
product(3,sum(2,4),2)=36
```

QUOTIENT

Returns the quotient of the supplied arguments.

Syntax

quotient(*argument* , *argument2…<argumentN>*)

Required Argument

argument*, *argument2*, … *argumentN
specifies a numeric value or a function that returns a numeric value.

Example

```
quotient(3,sum(2,4),2)=0.25
```

RANDOM

Returns a random number between the first argument and the second.

Syntax

random(*argument* , *argument2*)

Required Argument

argument*, *argument2
specifies a numeric value.

Example

```
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.

Syntax

rgx(*regular_expression* , *string…<group>*)

Required Arguments

regular_expression
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

Optional Argument

group
specifies a numeric reference to the regular expression.

Example

```
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. If no match is found, returns -1.

Syntax

rgxIndex(*regular_expression* , *string…<stringN>*)

Required Arguments

regular_expression
specifies a regular expression or a reference to a regular expression in the function context.

string…stringN
specifies a string.

Example

```
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.

Syntax

rgxLastToken(*regular_expression1* , *regular_expression2…<index_value>*)

Required Arguments

regular_expression1
specifies a regular expression or a reference to a regular expression in the function context.

regular_expression2
specifies a regular expression.

Optional Argument

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.

Example

```
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.

Syntax

rgxMatch(*regular_expression1* , *string…<group>*)

Required Arguments

regular_expression1
specifies a regular expression or a reference to a regular expression in the function context.

string
specifies a string.

Optional Argument

group
specifies a numeric reference to the regular expression. When specified, the result is the content of the specified numeric regular expression group.

Example

```
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.

Syntax

rgxReplace(*regular_expression1* , *string1*, *string2*)

Required Arguments

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.

Example

```
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.

Syntax

rgxReplaceAll(*regular_expression1* , *string1*, *string2*)

Required Arguments

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.

Example

```
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.

Syntax

rgxToken(*delimiter* , *string*, *<index>*)

Required Arguments

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. If 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.

Example

```
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.

Syntax

rgxV(*regular_expression* , *string*, *delimiter<group>*)

Required Arguments

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*.

Optional Argument

group
specifies a numeric reference to the regular expression. When specified, the result is the content of the specified numeric regular expression group.

Example

```
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.

Syntax

round(*argument*)

Required Argument

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.

Example

```
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.

Syntax

setContains(*set_of_tokens* , *delimiter*, *string*)

setContains(*#reference string*)

Required Arguments

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.

Example

```
setContains('one,two,three,four',',','two')=1
setContains(#mySet,'five')=0
```

STARTSWITH

Returns true if the first argument starts with the second. Otherwise, returns false.

Syntax

startsWith(*argument1* , *argument2*)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
startsWith('www.sas.com','www.')=1
startsWith('www.sas.com','sww.')=0
```

STRING

Returns the string value of the argument.

Syntax

string(*argument*)

Required Argument

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.

Example

```
string(33.9)=33.9
```

STRINGLENGTH

Returns the length of the string value of the argument.

Syntax

stringLength(*argument*)

Required Argument

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.

Example

```
stringLength('SAS ESP XML')=11
```

STRIP

Returns the string created after removing leading or trailing white space from the argument.

Syntax

strip(*argument*)

Required Argument

argument
specifies a string.

Example

```
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.

Syntax

substring(*argument* , *index*, <*length*>)

Required Arguments

argument
specifies a string.

index
specifies a numeric value that defines an index with which to parse the *argument*.

Optional Argument

length
specifies a numeric value. If specified, the substring is *length* size, otherwise it contains all characters to the end of the string.

Example

```
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.

Syntax

substringAfter(*argument1* , *argument2*, <*index*>)

Required Argument

argument1*, *argument2
specifies a string.

Optional Argument

index
specifies a numeric value that defines an index with which to parse *argument1*. When specified, the content after that occurrence is returned.

Example

```
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.

Syntax

substringBefore(*argument1* , *argument2*, <*index*>)

Required Argument

argument1*, *argument2
specifies a string.

Optional Argument

index
specifies a numeric value that defines an index with which to parse *argument1*. When specified, the content after that occurrence is returned.

Example

```
substringBefore('www.sas.com','.')=www
substringBefore('www.sas.com','.',2)=www.sas
```

SUM

Returns the sum of the numeric values of all arguments.

Syntax

sum(*argument* , *argument2*…<*argumentN*>)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
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.

Syntax

switch(*argument* , *argument2*, …<*argumentN*>, <*argumentN+1*>)

Required Argument

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 would specify this: **`#myXML`**.
- a function.

Example

```
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.

Syntax

systemMicro()

Example

```
systemMicro()=1420557039483912
```

SYSTEMMILLI

Returns the number of milliseconds since Jan 1, 1970.

Syntax

systemMilli()

Example

```
systemMilli()=1420557039483
```

TIMECURRENT

Returns the current time.

Syntax

timeCurrent()

Example

```
timeCurrent()=1421157236
timeString(timeCurrent())=Tue Jan 13 08:53:56 2015
```

TIMEDAYOFMONTH

Returns the day of the month of the current or specified time.

Syntax

timeDayOfMonth(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

timeDayOfWeek(<*argument*>)

Optional Argument

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).

Example

```
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.

Syntax

timeDayOfYear(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

timeGmtToLocal(*argument*)

Required Argument

argument
specifies a time value or a function that returns a time value.

Example

```
timeString(timeGmtToLocal(timeCurrent()))=Tue Jan 13 02:10:08 2015
```

TIMEGMTSTRING

Outputs the GMT time represented by the first argument.

Syntax

timeGmtString(*argument*, <*argument2*>)

Required Argument

argument
specifies a time value or a function that returns a time value.

Optional Argument

argument2
specifies a time format.

Example

```
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.

Syntax

timeHour(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
timeHour()=9
timeHour(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=14
```

TIMEMINUTE

Returns the minute of the hour of the current or specified time.

Syntax

timeMinute(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

timeMinuteOfDay(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

timeParse(*time*,<*format*>)

Required Argument

time
specifies a time specification or a function that returns a time specification.

Optional Argument

format
specifies a time format that is supported by the UNIX **`strftime`** function.

Example

```
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.

Syntax

timeSecond(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

timeStamp(<*format*>)

Optional Argument

format
specifies a time format that is supported by the UNIX **`strftime`** function.

Example

```
timeStamp()=Thu Feb 19 15:09:35 2015
timeStamp('%m-%d-%Y')=02-19-2015
```

TIMESTRING

Returns the time represented by the first argument.

Syntax

timeString(*time*,<*format*>)

Required Argument

time
specifies a time specification or a function that returns a time specification.

Optional Argument

format
specifies a time format. If you do not specify *format*, the system default time format is used.

Example

```
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.

Syntax

timeSecondofDay(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

timeToday()

Example

```
timeToday()=1421125200
```

TIMEYEAR

Returns the number of years since 1900 of the current or specified time.

Syntax

timeYear(<*argument*>)

Optional Argument

argument
specifies an expression that defines a specific time, or a function that returns a specific time.

Example

```
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.

Syntax

toLower(*string*)

Required Argument

string
specifies a string.

Example

```
toLower('Http://Www.Sas.Com/Products/Esp')=http://www.sas.com/products/esp
```

TOUPPER

Converts the value of the argument to uppercase.

Syntax

toUpper(*string*)

Required Argument

string
specifies a string.

Example

```
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.

Syntax

translate(*argument1*, *argument1*, *argument3*)

Required Argument

argument1*, *argument2*, *argument3
specifies a string. The length of *argument2* and *argument3* must be identical.

Example

```
translate('replace all vowels with its capital equivalent','aeiou','AEIOU')
          =rEplAcE All vOwEls wIth Its cApItAl EqUIvAlEnt
```

TRUE

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

Syntax

true(*argument*)

Required Argument

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.

Example

```
true('testing')=1
true('')=0
true(gt(10,5))=1
```

URLDECODE

Decodes the URL represented by the argument.

Syntax

urlDecode(*argument*)

Required Argument

argument
specifies a string.

Details

For more information to encoding and decoding URLs, see this reference.

Example

```
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.

Syntax

urlEncode(*argument*)

Required Argument

argument
specifies a string.

Details

For more information to encoding and decoding URLs, see this reference.

Example

```
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.

Syntax

xpath(*argument1*, *argument2*, <*argument3*>)

Required Arguments

argument1
specifies an instance of XML, represented by valid XML textual context or by a reference to XML elsewhere.

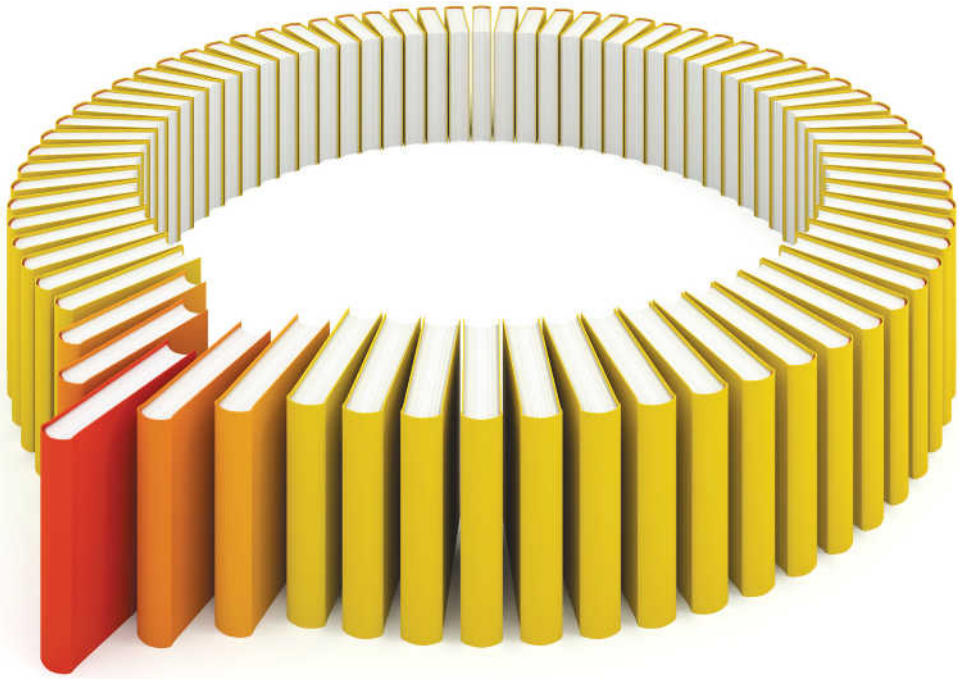
argument2
specifies an evaluation string.

Optional Argument

argument3
specifies a separator used when the function returns multiple results.

Example

```
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
```



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.