SAS® Event Stream Processing 4.2: Creating and Using Windows

Overview

An event stream processing model specifies how input event streams are transformed and analyzed into meaningful resultant event streams. Every model contains an engine. An engine contains one or more projects, and each project contains one or more continuous queries. A continuous query contains one or more*source windows* and one or more *derived windows*. Windows are connected by *edges*, which have an associated direction.

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

Using Expressions

Overview to Expressions

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

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

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

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

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

For information about how to specify DataFlux expressions, refer to the *DataFlux Expression Language: Reference Guide*. SAS Event Stream Processing uses a subset of the documented functionality, but this subset is robust for the needs of event stream processing.

Expression engine instances run window and splitter expressions for each event that is processed by the window. You can initialize expression engines before they are used by expression windows or window splitters (that is, before any events stream into those windows). Each expression window and window splitter has its own expression engine instance. Expression engine initialization can be useful to declare and initialize expression engine variables used in expression window or window splitter expressions. They can also be useful to declare regular expressions used in expressions.

To initialize expression engines for expression windows, use `dfESPexpression_window::expEngInitializeExp()`. To initialize expression engines for window splitters, use `dfESPwindow::setSplitter()`. You can find examples for both types of initialization in the event stream processing installs in **`$DFESP_HOME/examples/cxx`**. The expression window example is named splitter_with_initexp, and the window splitter example is named regex.

Understanding Data Type Mappings

An exact data type mapping does not exist between the data types supported by the SAS Event Stream Processing API and those supported by the DataFlux Expression Engine Language.

The following table shows the supported data type mappings.

Expression Data Type Mappings Table

Event Stream Processing Expressions	DataFlux Expressions	Notes and Restrictions
String (utf8)	String (utf8)	None
date (second granularity)	date (second granularity)	Seconds granularity
timestamp (microsecond granularity)	date (second granularity)	Constant milliseconds in dfExpressions not supported
Int32 (32 bit)	Integer (64 bit)	64-bit conversion for dfExpressions
Int64 (64 bit)	Integer (64 bit)	64-bit, no conversion
double (64 bit IEEE)	real (192 bit fixed decimal)	real 192-bit fixed point, double 64-bit float
money (192 bit fixed decimal)	real (192 bit fixed decimal)	192-bit fixed point, no conversion

Using Event Metadata in Expressions

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

Reserved Word	Opcode
ESP_OPCODE Use this reserved word to obtain the opcode of an event in a given window.	i — Insert u — Update p — Upsert d — Delete sd — Safe Delete A safe delete does not generate a "key not found" error.
ESP_FLAGS Use this reserved word in expressions to get the flags of an event in a given window.	N — Normal P — Partial R — Retention Delete

For an example, see *SAS Event Stream Processing 4.1: Examples*.

Using DataFlux Expression Language Global Functions

The DataFlux Expression Language supports global functions, also called user-defined functions (UDFs). You can register them as global functions and reference them from any expression window or window splitter expression. For more information about global functions, see *DataFlux Expression Language: Reference Guide*.

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

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

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

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

For an example, see *SAS Event Stream Processing 4.1: Examples* .

Using Blue Fusion Functions

Event stream processing expressions support the use of the DataFlux Data Management Platform quality functions (Blue Fusion Functions) . The following functions are fully documented in the *DataFlux Expression Language: Reference Guide*:

- bluefusion.case
- bluefusion.gender
- bluefusion.getlasterror

- bluefusion.identify
- bluefusion_initialize
- bluefusion.loadqkb
- bluefusion.matchcode
- bluefusion.matchscore
- bluefusion.pattern
- bluefusion.standardize

To use these functions, you must separately order and download the SAS DataFlux QKB (Quality Knowledge Base). You must set two environment variables as follows:

- `DFESP_QKB` to the share folder under the SAS DataFlux QKB installation. After you have installed SAS DataFlux QKB on Linux systems, this share folder is ***/QKB_root*/data/ci/*esp_version_number_no_dots*** (for example, **/QKB/data/ci/22**).
- `DFESP_QKB_LIC` to the full file pathname of the SAS DataFlux QKB license.

After you set up SAS DataFlux QKB for SAS Event Stream Processing, you can include these functions in any of your SAS event stream processing expressions. These functions are typically used to normalize event fields in the non-key field calculation expressions in a compute window.

For an example, see *SAS Event Stream Processing 4.1: Examples* .

Creating Source Windows

Overview to Source Windows

Source windows are required for each continuous query. All event streams enter continuous queries by being published or injected into a source window. Event streams cannot be published or injected into any other window type.

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

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

Defining Event Index Types

Source windows can have a primary index and a retention index. There are five stateful index types and one stateless index type. The primary index type of a source window affects the performance of the rest of the project. For example, when a source window has index type `pi_RBTREE`, the window is stateful and retains all incoming events. Retaining events at the source window without a retention policy uses substantially more memory as data is read into the continuous query than when the source window is stateless.

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

Retention Policies in Source Windows

You can define a retention policy in stateful source and copy windows. To use retention policies, the window cannot be specified as insert-only and the index type cannot be specified as `pi_EMPTY`. Retention policies limit

the flow of incoming data by time or event count, and thus can improve model performance. Events are deleted automatically by the engine when they exceed the window's retention policy. Usually, you want to follow any insert-only source window with a copy window with a retention policy.

For more information about retention policies, see Understanding Retention

Propagation of Insert-Only Processing

You can flag a source window to accept only Inserts. This attribute propagates to derived windows to until one of the following conditions is met:

- A window is determined not to produce insert-only data (for example, a window with retention).
- A window produces opcodes that cannot be determined from a graph analysis.

Automatically Generating Key Values

Source windows can automatically generate identification keys for incoming events. To automatically generate key values, the source window must be insert-only and have only one key with type INT64 or STRING. For INT64 keys, the value is an incremental count (0, 1, 2, ...). For STRING keys, the value is a Globally Unique Identifier (GUID).

The Publisher Connector

The publish connector is unique to the source window. The publish connector of a source window determines the following:

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

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

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

Examples of Source Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Using Metering Windows

Every event stream processing engine can maintain a metering window that tracks the number of events processed (and related timestamps) by all of the source windows in a model. Source windows inject events into the metering window at a default interval of five seconds. Subscribing applications can subscribe to the metering window as they do for any other window.

To enable this functionality, you must define the metafield `enableMetaProject` with the value **`true`** through the **`-meta`** argument to the XML server command or through the C++ API.

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

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

The metering window is contained in a continuous query named _meta_, which is contained in a project named _meta_. The project and query and window are created and started automatically when an engine is initialized.

An attempt to inject an event block into the metering window from anything other than a source window is rejected. A warning message is logged to the console. When a project is stopped, the number of events received since the last metering event is immediately sent to the metering window.

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

For information about how to define those additional fields, see "Starting and Using the Server" in *SAS Event Stream Processing: Using the XML Layer* .

Creating Copy Windows

Overview to Copy Windows

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

Retention Policies in Copy Windows

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

For more information about retention policies, see Understanding Retention

Examples of Copy Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Aggregate Windows

Overview to Aggregate Windows

Aggregate windows are similar to compute windows in that non-key fields are computed. However, key fields are specified, and not inherited from the input window. Key fields must correspond to existing fields in the input event. Incoming events are placed into aggregate groups with each event in a group that has identical values for the specified key fields.

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

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

Suppose that you specify the schema for an aggregate window as follows:

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

When events arrive in the aggregate window, they are placed into aggregate groups based on the value of the `symbol` field. Aggregate field calculation functions (written in C++) or expressions that are registered to the

aggregate window must appear in the non-key fields, in this example `totalQuant` and `maxPrice`. Either expressions or functions must be used for all of the non-key fields. They cannot be mixed. The functions or expressions are called with a group of events as one of their arguments every time a new event comes in and modifies one or more groups.

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

Flow of Operations

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

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

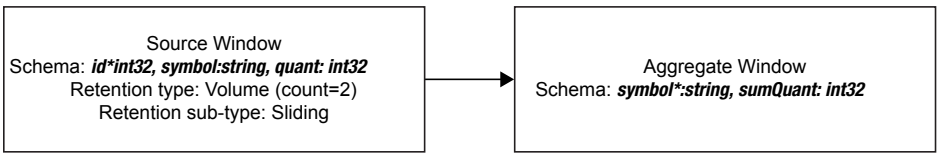
Using Aggregate Functions

Overview to Using Aggregate Functions

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

In a sense, the aggregate window partitions input events according to its keys. These partitions are called aggregate groups. The functions that are specified for non-key fields of the aggregate window are special functions. They operate on groups of values and collapse the group to a single scalar value.

Example of Aggregation



The key of the aggregate window is **`symbol`**. It has only one non-key field, **`sumQuant`**, which is the sum of the field **`quant`** arriving from the source window.

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

Aggregate Functions for Aggregate Window Field Calculation Expressions

The following aggregate functions are available for aggregate window field calculation expressions:

Aggregate Function	Returns
`ESP_aAve(fieldname)`	average of the group
`ESP_aCount()`	number of events in the group
`ESP_aCountDistinct(fieldname)`	the number of distinct non-null values in the column specified by field name within a group
`ESP_aCountNonNull(fieldname)`	the number of events with non-null values in the column for the specified field name within the group
`ESP_aCountNull(fieldname)`	the number of events with null values in the column for the specified field name within the group
`ESP_aCountOpcodes(opcode)`	count of the number of events matching opcode for group
`ESP_aFirst(fieldname)`	the first event added to the group
`ESP_aGUID()`	a unique identifier
`ESP_aLag(fieldname,lag_value)`	a lag value where the following holds: ■ `ESP_aLag(fieldname,0) == ESPaLast(fieldname)` ■ `ESP_aLag(fieldname,1)` returns the second lag. This is the previous value of *fieldname* that affected the group.
`ESP_aLast(fieldname)`	field from the last record that affected the group
`ESP_aLastNonNull(fieldname)`	field from the last record with non null value that affected the group
`ESP_aLastOpcodes(opcode)`	the opcode of the last record to affect the group
`ESP_aMax(fieldname)`	maximum of the group
`ESP_aMin(fieldname)`	minimum of the group
`ESP_aMode(fieldname)`	mode, or most popular of the group
`ESP_aStd(fieldname)`	standard deviation of the group
`ESP_aSum(fieldname)`	sum of the group
`ESP_aWAve(weight_fieldname, payload_fieldname)`	weighted group average

The following functions are always additive:

- `ESP_aAve`
- `ESP_aCount`
- `ESP_aCountOpcodes`

- ESP_aCountDistinct
- ESP_aGUID
- ESP_aLastOpcode
- ESP_aMode
- ESP_aStd
- ESP_aSum
- ESP_aWAve

The following functions are additive only when they get Inserts:

- ESP_aCountNonNull
- ESP_aCountNull
- ESP_aFirst
- ESP_aLast
- ESP_aLastNonNull
- ESP_aMax
- ESP_aMin

You can easily use the built-in aggregate functions for non-key field calculation expressions as follows:

```
dfESPwindow_aggregate *aw_01;
       aw_01 = cq->newWindow_aggregate("aggregateWindow_01",
                                       dfESPindextypes::pi_RBTREE,
                                       aggr_schema);
       aw_01->addNonKeyFieldCalc("ESP_aSum(quantity)"); // sum(quantity)
       aw_01->addNonKeyFieldCalc("ESP_aMax(quantity)"); // max(quantity)
```

Using aggregate field expressions is simpler than aggregate functions, but they perform slower, and the number of functions is limited.

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

The functions ESP_aSum, ESP_aFirst, ESP_aWAve, ESP_aStd, ESP_aCount, ESP_aLast, ESP_aFirst, ESP_aLastNonDelete, ESP_aLastOpCode, ESP_aCountOpcodes are all additive. That is, they can be calculated from retained state and the values determined from the incoming event. They do not need to maintain a group state. This means that if these are the only functions used in a dfESPwindow_aggrgate instance, special optimizations are made and speed-ups of an order of magnitude in the aggregate window processing can occur.

The dfESPgroupstate class is used internally to maintain the groups in an aggregation and an instance of the dfESPgroupstate is passed to aggregate functions. The signature of an aggregate function is as follows:

```
typedef dfESPdatavarPtr (*dfESPaggregate_func)(dfESPschema *is,
                                   dfESPeventPtr nep, dfESPeventPtr oep,
                                   dfESPgroupstate *gs);
```

You can find this definition in the **api/dfESPfuncptr.h** file.

The dfESPgroupstate object does not only act as a container for a set of events belonging to the same group, but it also maintains a state vector of dfESPdatavars, one state vector per non-key field, that can be used by aggregate functions to store a field's state. This enables quick incremental aggregate function updates when a new group member arrives.

Using an Aggregate Function to Add Statistics to an Incoming Event

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

For example, suppose that this is the incoming event schema:

```
"ID*:int64,symbol:string,time:datetime,price:double"
```

Suppose that with this incoming event schema, you want to add an aggregate statistic:

```
"ID*:int64,symbol:string,time:datetime,price:double,ave_price:double"
```

There, the average is calculated over the group with the same “symbol.”

Alternatively, you can define a single aggregate stream, with the following schema:

```
"ID:int64,symbol*:string,time:datetime,price:double,ave_price:double"
```

Note: The group-by is the key of the aggregation, which is "symbol".

Next, use `dfESPwindow_aggregate::addNonKeyFieldCalc(`*`expression`*`)` to register the following aggregation functions for each non-key field of this window, which in this case are “ID,” “time,” “price,” and “ave_price”:

```
awPtr->addNonKeyFieldCalc("ESP_aLast(ID)”);
awPtr->addNonKeyFieldCalc("ESP_aLast(time)”);
awPtr->addNonKeyFieldCalc("ESP_aLast(price)”);
awPtr->addNonKeyFieldCalc("ESP_aAve(price)”);
```

Suppose that the following events come into the aggregate window:

```
insert: 1, "ibm", 09/13/2001T10:48:00, 100.00
insert: 2, "orc", 09/13/2001T10:48:01, 127.00
insert: 3, "ibm", 09/13/2001T10:48:02, 102.00
insert: 4, "orc", 09/13/2001T10:48:03, 125.00
insert: 5, "orc", 09/13/2001T10:48:04, 126.00
```

The aggregate stream produces the following:

```
insert: 1, "ibm", 09/13/2001T10:48:00, 100.00, 100.00
insert: 2, "orc", 09/13/2001T10:48:01, 127.00, 127.00
update: 3, "ibm", 09/13/2001T10:48:00, 102.00, 101.00
update: 4, "orc", 09/13/2001T10:48:01, 125.00, 126.00
update: 5, "orc", 09/13/2001T10:48:01, 126.00, 126.00
```

By using `aLast(`*`fieldname`*`)` and then adding the aggregate fields of interest, you can avoid the subsequent join window. This makes the modeling cleaner.

Writing and Using an Aggregate Function

Write aggregate functions with zero, one, two or three arguments. The arguments must be either integer-valued DataFlux expressions, integer constants, or field names in the input schema for the aggregation.

The most commonly used aggregate functions are one parameter functions with an input schema field name (for example, the built-in aggregation function `ESP_aMax(`*`fieldname`*`)`). For field names in the input schema, the field index into the input event is passed into the aggregate function, not the value of the field. This is important when you deal with groups. You might need to iterate over all events in the group and extract the values by event index from each input event.

After you write an aggregate function, embed it in C++ code in order to use it in your event stream processing application. How to do this is documented in an example provided in **$DFESP_HOME/examples/cxx/aggregate_userdef**.

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

```
// the uMyFunction wrapper:
     // every aggregation function must be wrapped like this.
    //
    int dfESPaggrfunc_uMyFunctionWrapper(dfESPexpEngine::exp_engine_t *e,
                                         dfESPexpEngine::exp_sym_value_t *returnval,
                                         int parmcount,
                                         dfESP_EXPsym_value_t **parms)
                                          {
        return dfESPaggrfunc_Wrapper((void *)my_aggrergation_function, e,
                                            returnval, parmcount, parms);
                                          }
```

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

```
// SAS Event Stream Processing calls this function to get all user-defined
    //    aggregation functions during the initialization stage.
    //
    void add_user_aggrFunctions() {
        dfESPengine *e = dfESPengine::getEngine();

        dfESPptrList<aggr_function_t *> &uFuncts = e->getUDAFs();

        // push back as many user defined functions as you like:
        // the parameters are: <callable name>, <function pointer>,
        // <num args>, <additive flag>, <additive flag for insert only>,
        // <description>
        uFuncts.push_back(new aggr_function_t("USER_myFunction",
            (void *)dfESPaggrfunc_uMyFunctionWrapper,
                    1, true, true, "description"));
    }
```

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

Writing Non-Additive Aggregate Functions

The simplest aggregate sum function does not maintain state and is not additive. The function iterates through each event in a group to aggregate. It requires the aggregation window to maintain a copy of every input event for all groups.

The following code performs these basic steps:

1. Look at the input and output types and verify compatibility.
2. Initialize a return variable of the specified output type.
3. Loop across all events in the group and perform the aggregation function.
4. Check for computational errors and return the error or the result .

```
// a non-additive summation function
    //
    // vgs is the groupstate object passed as a (void *) pointer
    // fID is the filed ID in internal field order of the field on
    //    which we sum.
    dfESPdatavarPtr uSum_nadd(void *vgs, size_t fID) {

        dfESPdatavar    *rdv;
        // placeholder for return value
        dfESPgroupstate *gs = (dfESPgroupstate *)vgs;
        // the passed groupstate cast back to dfESPgroupstate object.

        // get the 1) aggregate schema (output schema)
        //     and 2) the schema of input events
        //
        dfESPschema  *aSchema = gs->getAggregateSchema();
        dfESPschema  *iSchema = gs->getInputSchema();

        // get the type of 1) the field we are computing in the aggregate schema
        // and 2) the input field we are summing.
        //
        dfESPdatavar::dfESPdatatype aType =
          aSchema->getTypeEO(gs->getOperField());
        dfESPdatavar::dfESPdatatype iType =
          iSchema->getTypeIO(fID);

        dvn_error_t   retCode = dvn_noError;
        // return code for using the datavar numerics package.

        // If the input fields or the output field is non-numeric,
        // flag an error.
        //
        if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
        cerr << "summation must work on numeric input, produce numeric output."
             << endl;
        return NULL;
        }

        // in the ESP type system, INT32 < INT64 < DOUBLE < DECSECT.
        //   This checks compatibility. The output type must be greater
        //   equal the input type. i.e. we cannot sum a column of int64
        //   and puit them into an int32 variable.
        //
        if (iType > aType) {
         cerr << "output type is not precise enough for input type" << endl;
         return NULL;
        }

        dfESPeventPtr nev = gs->getNewEvent();
        dfESPeventPtr oev = gs->getOldEvent();

        // create the datavar to return, of the output type and set to zero.
        //
        rdv = new dfESPdatavar(aType);     // NULL by default.
        rdv->makeZero();
```

```
dfESPeventPtr gEv = gs->getFirst(); // get the first event in the group.
dfESPdatavar iNdv(iType);           // a place to hold the input variable.
while (gEv) {                       // iterarate until no more events.
    gEv->copyByIntID(fID, &iNdv);   // extract value from record into iNdv;

    if (!iNdv.isNull()) {           // input not null
        if ((retCode = dv_add(rdv, rdv, &iNdv)) != dvn_noError)
            break;                  // rdv = add(rdv, iNdv)
    }
    gEv = gs->getNext();            // get the first event in the group.
}

if (retCode != dvn_noError ) {       // if any of our arithmitic fails.
 rdv->null();                        // return a null value.
 cerr << "uSum() got an arithmetic error in summing up values" << endl;
}

return rdv;
```

Writing Additive Aggregate Functions

Aggregate functions that compute themselves based on previous field state and a new field value are called additive aggregation functions. These functions provide computational advantages over aggregate functions.

An additive aggregate function can be complex for two reasons:

- They must look at the current state (for example, the last computed state).
- They must evaluate the type of incoming event to make proper adjustments.

Suppose that you keep the state of the last value of the group's summation of a field. When a new event arrives, you can conditionally adjust the state base on whether the incoming event is an Insert, Delete, or Update. For an Insert event, you simply increase the state by the new value. For a Delete, you decrease the state by the deleted value. For an Update, you increase and decrease by the new and old values respectively. Now the function never has to loop through all group values. It can determine the new sum based on the previous state and the latest event that affects the group.

The following code performs these basic steps:

1. Look at the input and output types and verify compatibility.
2. Initialize a return variable of the specified output type.
3. Determine whether the function has been called before. That is, is there a previous state value?
   - If so, retrieve it for use.
   - If not, create a new group with an arriving insert so that you can set the state to the incoming value.
4. Switch on the opcode and adjust the state value.
5. Check for computational errors and return the error value or the state value as the result.

```
// an additive summation function
    //
    // vgs is the groupstate object passed as a (void *) pointer
    // fID is the filed ID in internal field order of the field on
    //    which we sum.
    dfESPdatavarPtr uSum_add(void *vgs, size_t fID) {
```

```
dfESPdatavar    *rdv;
// placeholder for return value
dfESPgroupstate *gs = (dfESPgroupstate *)vgs;
// the passed groupstate cast back to dfESPgroupstate object.

// get the 1) aggregate schema (output schema)
// and 2) the schema of input events
//
dfESPschema  *aSchema = gs->getAggregateSchema();
dfESPschema  *iSchema = gs->getInputSchema();

// get the type of 1) the field we are computing in the aggregate schema
// and 2) the input field we are summing.
//
dfESPdatavar::dfESPdatatype aType =
  aSchema->getTypeEO(gs->getOperField());
dfESPdatavar::dfESPdatatype iType =
  iSchema->getTypeIO(fID);


dvn_error_t   retCode = dvn_noError;
// return code for using the datavar numerics package.

// If the input fields or the output field is non-numeric,
// flag an error.
//
if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
cerr << "summation must work on numeric input, produce numeric output."
     << endl;
return NULL;
}

// in the ESP type system, INT32 < INT64 < DOUBLE < DECSECT.
//   This checks compatibility. The output type must be greater
//   equal the input type. i.e. we cannot sum a column of int64
//   and puit them into an int32 variable.
//
if (iType > aType) {
cerr << "output type is not precise enough for input type" << endl;
return NULL;
}

// fetch the input event from the groupstate object (nev)
// and, in the case of an update, the old event that
// is being updated (oev)
//
dfESPeventPtr nev = gs->getNewEvent();
dfESPeventPtr oev = gs->getOldEvent();

// Get the new value out of the input record
//
dfESPdatavar iNdv(iType);
// a place to hold the input variable.
dfESPdatavar iOdv(iType);
// a place to hold the input variable (old in upd case).
```

```
nev->copyByIntID(fID, iNdv);
// extract input value (no copy) to it (from new record)

// Get the old value out of the input record (update)
//
if (oev) {
    oev->copyByIntID(fID, iOdv);
// extract input value to it (old record)
}

// Note: getStateVector() returns a reference to the state vector for
//       the field we are computing inside the group state object.
//
dfESPptrVect<dfESPdatavarPtr> &state = gs->getStateVector();

// create the datavar to return, of the output type and set to zero.
//
rdv = new dfESPdatavar(aType);     // NULL by default.
rdv->makeZero();

// If the state has never been set, we set it and return.
//
if (state.empty()) {
    dv_assign(rdv, &iNdv);
    //  result = input
    state.push_back(new dfESPdatavar(rdv));
    //  make a copy and push as state
    return rdv;
}

// at this point we have a state,
// so lets see how we should adjust it based on opcode.
//

dfESPeventcodes::dfESPeventopcodes opCode = nev->getOpcode();
bool  badOpcode = false;
int   c = 0;
switch (opCode)  {
case dfESPeventcodes::eo_INSERT:
    if (!iNdv.isNull())
        retCode = dv_add(state[0], state[0], &iNdv);
    break;
case dfESPeventcodes::eo_DELETE:
    if (!iNdv.isNull())
        retCode = dv_subtract(state[0], state[0], &iNdv);
    break;
case dfESPeventcodes::eo_UPDATEBLOCK:
    retCode = dv_compare(c, &iNdv, &iOdv);
    if (retCode != dvn_noError) break;
    if (c == 0)  // the field value did not change.
        break;
    if (!iNdv.isNull())
    // add in the update value
        retCode = dv_add(state[0], state[0], &iNdv);
    if (retCode != dvn_noError) break;
    if (!iOdv.isNull())
```

```
            // subtract out the old value
                retCode = dv_subtract(state[0], state[0], &iOdv);
            break;
        default:
        cerr << "got a bad opcode when running uSum_add()" << endl;
        badOpcode = true;
        }

        if ( badOpcode || (retCode != dvn_noError) ) {
            rdv->null();
            // return a null value.
        cerr << "uSum() got an arithmetic error summing up values" << endl;
        } else
            dv_assign(rdv, state[0]);
        // return the adjusted state value

        return rdv;
    }
```

Even though it is possible to use the `Sum()` aggregate function to iterate over the group and compute a new sum when a new group changes, faster results are obtained when you maintain the `Sum()` in a `dfESPdatavar` in the `dfESPgroupstate` object and increment or decrement the object by the incoming value, provided that the new event is an Insert, Update, or Delete. The function then adjusts this field state so that it is up-to-date and can be used again when another change to the group occurs.

Examples of Aggregate Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Compute Windows

Overview to Compute Windows

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

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

Using Compute Functions

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

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

User-defined functions are specified in the **`udf`** of the **`udfs`** and **`expr-initialize`** elements at the beginning of **`window-compute`**.

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

Examples of Compute Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Counter Windows

Overview to Counter Windows

A counter window enables you to see how many events are going through your model and the rate at which they are being processed.

The XML element to create a counter window is as follows:

```
<window-counter name='name' count-interval='period' clear-interval='cperiod'/>
```

Field	Description
`name`	The name of the window.
`count-interval` (optional)	Period of time for which to generate a current event processing rate, specified in `'[value] [unit]'`.
`clear-interval` (optional)	Period of inactivity after which all counter data is reset to 0, specified in `'[value] [unit]'`.

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

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

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

The opcode for generated events is based on the index of the counter window. If the index is `pi_EMPTY`, the opcode is Insert. For any other index value, the opcode is Upsert.

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

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

```
    </event>
```

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

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

To use a counter window, add an edge with the counter window as the target and the window to monitor as the source. You can connect multiple windows to the same counter window. Streamviewer can subscribe to the counter window to show the results. Alternatively, you can add the counter window to the trace attribute of the `<contquery>` element that prints formatted events to standard output.

Counter Window Examples

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Filter Windows

Overview to Filter Windows

Filter windows use registered Boolean filter functions or expressions to determine what inputs events are allowed into the filter window. These functions and expressions are called filter conditions.

Using Filter Windows

Filter conditions available to filter windows include expressions, user-defined functions (global functions), and registered plug-in functions.

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

Examples of Filter Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Functional Windows

Overview to Functional Windows

A functional window enables you to use different types of functions to manipulate or transform event data. You define a schema and then a *function context* that contains functions and supporting entities such as regular expressions, XML, and JSON. When an event enters a functional window, the window looks for a function with a name that corresponds to each field in its schema. If the function exists, it is run and the resulting value is entered into the output event. If no function is specified for a field, and a field with the same name exists in the input schema, the input value is copied directly into the output event.

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

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

Using Event Loops

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

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

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

Understanding and Using Function Context

Overview to Function Context

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

Types of Functions You Can Use

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

- general functions
- functions specific to event stream processing

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

Here are the data mappings relevant to using functions in a function context:

string	ESP_UTF8STR
float	ESP_DOUBLE
long	ESP_INT64
integer	ESP_INT32
Boolean	ESP_INT32

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

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

You can also reference fields in the output event. Continuing the above example, perhaps you want to add the `hourlyWage` field to the output event depending on the value of `occupation`:

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

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

Using Expressions

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

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

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

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

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

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

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

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

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

Specifying Properties

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

There are five types of properties:

- `map` executes the function to generate a map of name-value pairs to be used for value lookups by name

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

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

Here is how you would code each property type

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

Suppose you had some employee information streaming into the model.

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

You can create a `property-map` to store employee data and then examine the position field in order to create a `property-xml` containing the appropriate data. If the employee is a developer, the XML is from `developerInfo`. Otherwise, it uses `managerInfo`. Your function-context would look like this:

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

Streaming in the following event:

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

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

Yields the following result:

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

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

Function-Context Example

```
<function-context>
   <expressions>
      <expression name='myexp'>posix_regular_expression</expression>
      ...
   </expressions>
   <properties>
      <property-map name='map' outer=';' inner='='>code</property-map>
      <property-xml name='xmlprop'>code</property-xml>
      <property-json name='jsonprop'>code</property-json>
      <property-string name='string'>code</property-string>
      <property-set name='mySet' delimiter=','>code</property-set>
      ...
   </properties>
   <functions>
      <function name='mySum'>code</function>
      ...
   </functions>
</function-context>
```

XML Examples of Functional Windows

Stock Trades

Suppose that you had stock trade information streaming into an event stream processing model. You want to generate an event anytime that a huge trade (> 150000 shares) takes place during the first or last 15 minutes of the trading day.

Your model includes the following source window:

```
<window-source name='source' insert-only='true'>
   <schema>
      <fields>
         <field name='id' type='int32' key='true'/>
         <field name='symbol' type='string'/>
         <field name='currency' type='int32'/>
         <field name='time' type='int64'/>
         <field name='msecs' type='int32'/>
         <field name='price' type='double'/>
         <field name='quant' type='int32'/>
         <field name='venue' type='int32'/>
         <field name='broker' type='int32'/>
         <field name='buyer' type='int32'/>
         <field name='seller' type='int32'/>
         <field name='buysellflg' type='int32'/>
      </fields>
   </schema>
</window-source>
```

You stream events from the source window into a filter window to obtain huge trades:

```
<window-filter name='hugeTrades'>
   <expression><![CDATA[quant>150000]]></expression>
</window-filter>
```

That data flows into the functional window:

```
<window-functional name='transform'>
   <schema>
      <fields>
         <field name='id' type='int32' key='true'/>
         <field name='symbol' type='string'/>
         <field name='timeString' type='string'/>
         <field name='hourOfDay' type='double'/>
         <field name='quant' type='int32'/>
      </fields>
   </schema>
   <function-context>
      <functions>
         <function name='timeString'>
            timeString($time)
         </function>
         <function name='hourOfDay'>
            precision(quotient(timeSecondOfDay($time),3600),2)
          </function>
      </functions>
   </function-context>
</window-functional>
```

Several fields in are defined in the schema of the functional window. However, functions are defined only for a few fields. The remaining fields are copied from the input event.

You include functions to do the following things: format a time into a readable form in the `timeString` field, and then calculate a floating-point value representing the hour of the day on which the trade was made.

```
<function name='hourOfDay'>
          precision(quotient(timeSecondOfDay($time),3600),2)
       </function>
```

This function produces a floating-point value that you can stream to a filter in order to get early and late huge trades.

```
<window-filter name='earlyTrade'>
   <expression><![CDATA[hourOfDay<9.75]]></expression>
</window-filter>

<window-filter name='lateTrade'>
   <expression><![CDATA[hourOfDay>15.75]]></expression>
</window-filter>
```

Running several million trades through the model generates output like the following:

```
<event opcode='insert' window='project/query/earlyTrade'>
   <value name='hourOfDay'>9.73</value>
   <value name='id'>11847604</value>
   <value name='quant'>1000000</value>
   <value name='symbol'>NOK</value>
   <value name='timeString'>Wed Aug  4 09:43:48 2010</value>
</event>

<event opcode='insert' window='project/query/lateTrade'>
```

```
    <value name='hourOfDay'>15.81</value>
    <value name='id'>16739499</value>
    <value name='quant'>270400</value>
    <value name='symbol'>TXT</value>
    <value name='timeString'>Wed Aug  4 15:48:47 2010</value>
  </event>
```

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Join Windows

Overview to Join Windows

A join window takes two input windows and a join type. For example,

- left outer window
- right outer window
- inner join
- full outer join

A join window takes a set of join constraints and a non-key field signature string. It also takes one of the following for the calculation of the join non-key fields when new input events arrive:

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

A join window produces a single output stream of joined events. Because an engine is based on primary keys and supports Inserts, Updates, and Deletes, there are some restrictions placed on the types of joins that can be used.

The left window is the first window added as a connecting edge to the join window. The second window added as a connecting edge is the right window.

Understanding Streaming Joins

Overview to Streaming Joins

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

Join Type	Description
one-to-one joins	An event on either side of the join can match at most one event from the other side of the join. This type of join always involves two dimension tables.

Join Type	Description
one-to-many joins (or many-to-one joins)	An event that arrives on one side of the join can match many rows of the join. An event that arrives on the other side of the join can match at most one row of the join. There are two conditions for a join to be classified as a one-to-many (or many-to-one) join: ■ a change to one side of the join can affect many rows ■ a change to the other side can affect at most one row Both conditions must be met.
many-to-many joins	A single event that arrives on either side of the join can match more than one event on the other side of the join.

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

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

The join constraints are an n-tuple of equality expressions. Each expression involves one field from the left window and one field from the right. For example: *(left.*f_1 = = *right.*f_{10}*), (left.*f_2 = = *right.*f_7*), ... (left.field*$_{10}$ == *right.field*$_1$*)* .

In a streaming context, every window has a primary key that enables the insertion, deletion, and updating of events. The keys for a join window are derived from the total set of keys from the left window and the right window. When an event arrives on either side, you must be able to compute how the join changes, given the nature of the arriving data (Insert, Update, or Delete). The theory of join-key derivation that SAS Event Stream Processing follows maintains consistency for the most common join cases.

Some of the basic axioms used in the join-key derivation are as follows:

- For a left-outer join, the keys of the join cannot contain any keys from the right window. A joined event is output when a left event arrives. There is no matching event on the right.
- For a right-outer join, the keys of the join cannot contain any keys from the left window. A joined event is output when a right event arrives. There is no matching event on the left.
- For a many-to-many join, the keys of the joins need to be the union of the keys for the left and right windows. To understand this axiom, think of an event coming in on one side of the join that matches multiple events on the other side of the join. In this case, all the keys of the many side must be included. Otherwise, you cannot distinguish the produced events. Because the single event that matches many events can come on the other side of the join, reverse the above statement to determine what happens in a streaming context. All keys from the left and right side of the join must be present.
- For one-to-many or many-to-one joins, the side of the join that matches multiple events is the side from which that the join windows keys derive. This is the case when there is a single event on the other side.

Join windows are either dimension windows or fact windows. Dimension windows are those whose entire set of key fields participate in the join constraints. Fact windows are those that have at least one key field that does not participate in the join constraints.

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

Classification	Left Window	Right Window	Allowed Type	Key Derivation	Streaming Window
one-to-one	Dimension	Dimension	Left Outer	join keys are keys of left window	Left
			Right Outer	join keys are keys of right window	Right
			Full Outer	join keys are keys of left window (arbitrary choice)	Left
			Inner	join keys are keys of left window (arbitrary choice)	Left
one-to-many	Fact	Dimension	Left Outer	join keys are keys of left window (right window is lookup)	Left
			Inner	join keys are keys of left window (right window is lookup)	Left
many-to-one	Dimension	Fact	Right Outer	join keys are keys of right window (left window is lookup)	Right
			Inner	join keys are keys of right window (left window is lookup)	Right
many-to-many	Fact	Fact	Inner	join keys are the full set of keys from the left and right windows	None

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

Using Secondary Indices

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

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

When a large number of changes are possible to the dimension table, it is suggested to enable a secondary index on the join. Automatic secondary index generation is enabled by specifying a join parameter when you construct a new join window. This causes a secondary index to be generated and maintained automatically when the join type involves a dimension table. This has the advantage of eliminating all table scans when changes are made to the dimension table. There is a slight performance penalty when you run with secondary indices turned on. The index needs to be maintained with every update to the fact table. However, this

secondary index maintenance is insignificant compared with elimination of table scans. With large tables, you can achieve time savings of two to three orders of magnitude through the use of secondary indices.

For many-to-many joins, enabling on secondary indices is recommended.

Using Regeneration versus No Regeneration

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

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

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

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

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

Creating Empty Index Joins

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

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

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

- A many-to-many join cannot have an empty index.

- The streaming side of a join, as specified in the join classification table, can receive only inserts.

Examples of Join Windows

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

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

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

If `sw_01` is the window identifier for the left input window and `sw_02` is the window identifier for the right input window, your code would look like this:

```
dfESPwindow_join *jw;
jw = cq->newWindow_join("myJoinWindow", dfESPwindow_join::jt_LEFTOUTER,
                        dfESPindextypes::pi_RBTREE);
jw-> setJoinConditions ("l_ID==r_tID");
jw->setJoinSelections("l_symbol,l_price,l_traderID,r_name");
jw->setFieldSignatures("sym:string,price:double,tID:int32,
     traderName:string");
```

Note the following:

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

  ```
  "l_fieldname=r_fieldname, ...,l_fieldname=r_fieldname"
  ```

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

  ```
  "{l|r}_fieldname, ...{l|r}_fieldname"
  ```

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

  ```
  "fieldname:fieldtype, ..., fieldname:fieldtype
  ```

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

```
dfESPwindow_join *jw;
jw = cq->newWindow_join("myJoinWindow", dfESPwindow_join::jt_LEFTOUTER,
                        dfESPindextypes::pi_RBTREE);
jw->setJoinConditions("l_ID==r_tID");
jw->addNonKeyFieldCalc("l_symbol");
jw->addNonKeyFieldCalc("l_price");
jw->addNonKeyFieldCalc("l_traderID");
jw->addNonKeyFieldCalc("r_name");
jw->setFieldSignatures("sym:string,price:double,tID:int32,
     traderName:string");
```

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

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Notification Windows

Overview to Notification Windows

Notification windows enable you to send notifications through email (SMTP), text (SMS), and or multimedia message (MMS). These windows, like functional windows, enable you to define a function context to transform incoming events before processing them for possible notifications. Each of the different types of notification has its own configuration requirements. For example, an email requires that the configuration specify the event field that contains the 'send to' email address. SMS and MMS require phone numbers and phone provider gateway information.

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

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

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

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

The full XML configuration of a notification window is as follows:

```
<window-notification name=''>
   <schema>
   ...
   </schema>

   <function-context>...</function-context>

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

   <delivery-channels>

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

```
            </email-contents>
         </email>

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

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

      </delivery-channels>

   </window-notification>
```

Notification windows use Simple Mail Transfer Protocol (SMTP) to send email, Short Message Service (SMS), and Multimedia Messaging Service (MMS) messages. To use these delivery channels, you must specify an `smtp` element to provide information about an SMTP server:

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

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

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

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

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

Notification Window Delivery Channels

Overview to Notification Window Delivery Channels

The notification window uses three types of delivery channel:

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

Using the Email Delivery Channel

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

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

The `email` element contains the following attributes:

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

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

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

- the `sender` email address
- the `recipients` to whom the email message is sent
- the `subject` of the email
- the `from` text of the email message
- the `to` text of the email
- The `email-contents` element, which contains the following elements:
  - the `text-content` element encloses the plain text content of the message
  - the `html-content` element encloses the HTML content of the message
  - the `image-content` element encloses a URL to image data

These elements can be interspersed in any way you want. The content of each element is included in the message in the order in which it appears. Any image data is retrieved and `base64` encoded before being inserted into the message.

Using the SMS Delivery Channel

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

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

The `sms` element contains the following attributes:

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

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

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

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

Using the MMS Delivery Channel

Here is XML code to use the MMS delivery channel:

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

```
</mms>
```

The `mms` element contains the following attributes:

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

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

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

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

These elements can be interspersed in any way you want. The content of each element is included in the message in the order it appears. Any image data is retrieved and `base64` encoded before being inserted into the message.

Using the Function-Context Element

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Suppose you had employee information streaming into the model.

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

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

Specify the `function-context` element as follows:

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

When you stream the following events:

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

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

The `function-context` yields the following:

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

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

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

Examples of Notification Windows

Building a Streaming Performance Monitor

Suppose that you want to stream a counter window into a filter window in order to check whether total throughput rate has dropped below 130,000 events per second. When that condition occurs, the event streams into a notification window that sends an SMS text message alerting someone of the slow streaming condition.

Here is the counter window:

```
<window-counter name='counter'
                       count-interval='2 seconds'
                       clear-interval='30 seconds'/>
```

It feeds the following filter window:

```
<window-filter name='poorPerformance'>
   <expression><![CDATA[totalSeconds > 10 and totalRate<130000]]></expression>
</window-filter>
```

Note that the `totalSeconds > 10`. You anticipate clocking slower rates as the data begins to stream.

Next, feed the event that indicates poor performance into a notification window:

```
<window-notification name='reportPerformance'>
   <smtp host='mailhost.fyi.sas.com' />
   <delivery-channels>
      <sms test='false' throttle-interval='2 hours'>
         <sms-info>
            <sender>brenda.doe@orion.com</sender>
            <from>ESP_Trade_Monitor</from>
            <subject>Slow Streaming</subject>
            <gateway>txt.att.net</gateway>
            <phone>5556466705</phone>
         </sms-info>
         <sms-contents>
            <text-content>
                The trade streaming has become very slow.
                It is only processing $totalRate trades per second after running
                for $totalSeconds seconds.
            </text-content>
         </sms-contents>
      </sms>
```

```
      </delivery-channels>
   </window-notification>
```

You do not need extra schema or function context in this example. All the information you want to send is in the input event. The event generated by a counter window that looks like this:

```
<event opcode='upsert' window='project/query/counter'>
   <value name='input'>source</value>
   <value name='intervalCount'>283473</value>
   <value name='intervalRate'>141736</value>
   <value name='intervalSeconds'>2</value>
   <value name='totalCount'>782662</value>
   <value name='totalRate'>130444</value>
   <value name='totalSeconds'>6</value>
</event>
```

Grab the `totalRate` field and send it in an SMS text message along with the number of seconds that events have been streaming. The recipient gets an SMS text message with this data.

Catching Front Running Traders

The following example catches stock traders when they attempt front running buys. A broker caught in the act is sent an email, an SMS text message, and an MMS message. The message includes graphic details of the trades involved in the violation, and for the channels that permit graphics, contains an image of someone in a jail cell. All relevant message routing information is included in the broker dimension data:

```
i,n,1012112,Frodo,ESP,940 Orion Suite 201 Cary NC 27513,,frodo.doe@orion.com,5556466705,txt.att.net,mms.att.net
i,n,1012223,Sam,ESP,940 Orion Suite 201 Cary NC 27513,,sam.doe@orion.com,5556466706,txt.att.net,mms.att.net
i,n,1012445,Pippin,ESP,940 Orion Suite 201 Cary NC 27513,pippin.doe@orion.com,5556466707,txt.att.net,mms.att.net
i,n,1012334,Merry,ESP,940 Orion Suite 201 Cary NC 27513,merry.doe@orion.com,5556466708,txt.att.net,mms.att.net
i,n,101667,Gandalf,ESP,940 Orion Suite 201 Cary NC 27513,gandalf.doe@orion.com,5556466709,txt.att.net,mms.att.net
i,n,1012001,Aragorn,ESP,940 Orion Suite 201 Cary NC 27513,aragorn.doe@orion.com,5556466710,txt.att.net,mms.att.net
```

Note that the last four fields contain the email, phone number, and SMS and MMS gateways for each broker.

First, data streams into the model through a source window.

```
<window-source name='brokersSource' insert-only='true'>
   <schema-string>broker*:int32,brokerName:string,brokerage:string,
                  brokerAddress:string,brokerPhone:string,email:string,
                  smsGateway:string,mmsGateway:string</schema-string>
   <connectors>
      <connector class='fs'>
         <properties>
            <property name='type'>pub</property>
            <property name='fstype'>csv</property>
            <property name='fsname'>data/brokers.csv</property>
         </properties>
      </connector>
   </connectors>
</window-source>
```

A pattern window is constructed to detect front running violations. The pattern window needs to carry a lot of information because it deals with up to three trades. Each trade contains broker and customer information as well as the trade data. All of this data must be available to format a notification message.

The pattern window looks like this:

```
<window-pattern name='frontRunning'>
   <schema>
      <fields>
```

```
      <field name='id' type='int64' key='true' />
      <field name='broker' type='int32' />
      <field name='brokerName' type='string' />
      <field name='email' type='string' />
      <field name='phone' type='string' />
      <field name='sms' type='string' />
      <field name='mms' type='string' />
      <field name='customer' type='int32' />
      <field name='symbol' type='string' />
      <field name='tstamp1' type='string' />
      <field name='tstamp2' type='string' />
      <field name='tstamp3' type='string' />
      <field name='tradeId1' type='int32' />
      <field name='tradeId2' type='int32' />
      <field name='tradeId3' type='int32' />
      <field name='price1' type='double' />
      <field name='price2' type='double' />
      <field name='price3' type='double' />
      <field name='quant1' type='int32' />
      <field name='quant2' type='int32' />
      <field name='quant3' type='int32' />
      <field name='slot' type='int32' />
   </fields>
</schema>
<splitter-expr>
   <expression>slot</expression>
</splitter-expr>
<patterns>
   <pattern index='broker,symbol'>
      <events>
         <event name='e1'>((buysellflg == 1)
          and (broker == buyer)
          and (s == symbol)
          and (b == broker)
          and (p == price))</event>
         <event name='e2'>((buysellflg == 1)
          and (broker != buyer)
          and (s == symbol)
          and (b == broker))</event>
         <event name='e3'><![CDATA[((buysellflg == 0)
          and (broker == seller)
          and (s == symbol)
          and (b == broker)
          and (p < price))]]></event>
      </events>
      <logic>fby{1 hour}(fby{1 hour}(e1,e2),e3)</logic>
      <output>
         <field-selection name='broker' node='e1'/>
         <field-selection name='brokerName' node='e1'/>
         <field-selection name='brokerEmail' node='e1'/>
         <field-selection name='brokerPhone' node='e1'/>
         <field-selection name='brokerSms' node='e1'/>
         <field-selection name='brokerMms' node='e1'/>
         <field-selection name='buyer' node='e2'/>
         <field-selection name='symbol' node='e1'/>
         <field-selection name='date' node='e1'/>
```

```
            <field-selection name='date' node='e2'/>
            <field-selection name='date' node='e3'/>
            <field-selection name='id' node='e1'/>
            <field-selection name='id' node='e2'/>
            <field-selection name='id' node='e3'/>
            <field-selection name='price' node='e1'/>
            <field-selection name='price' node='e2'/>
            <field-selection name='price' node='e3'/>
            <field-selection name='quant' node='e1'/>
            <field-selection name='quant' node='e2'/>
            <field-selection name='quant' node='e3'/>
            <field-expr>1</field-expr>
         </output>
      </pattern>
      <pattern index='broker,symbol'>
         <events>
            <event name='e1'>((buysellflg == 0)
             and (broker == seller)
             and (s == symbol)
             and (b == broker))</event>
            <event name='e2'>((buysellflg == 0)
             and (broker != seller)
             and (s == symbol)
             and (b == broker))</event>
         </events>
         <logic>fby{10 minutes}(e1,e2)</logic>
         <output>
            <field-selection name='broker' node='e1'/>
            <field-selection name='brokerName' node='e1'/>
            <field-selection name='brokerEmail' node='e1'/>
            <field-selection name='brokerPhone' node='e1'/>
            <field-selection name='brokerSms' node='e1'/>
            <field-selection name='brokerMms' node='e1'/>
            <field-selection name='seller' node='e2'/>
            <field-selection name='symbol' node='e1'/>
            <field-selection name='date' node='e1'/>
            <field-selection name='date' node='e2'/>
            <field-expr> </field-expr>
            <field-selection name='id' node='e1'/>
            <field-selection name='id' node='e2'/>
            <field-expr>0</field-expr>
            <field-selection name='price' node='e1'/>
            <field-selection name='price' node='e2'/>
            <field-expr>0</field-expr>
            <field-selection name='quant' node='e1'/>
            <field-selection name='quant' node='e2'/>
            <field-expr>0</field-expr>
            <field-expr>2</field-expr>
         </output>
      </pattern>
   </patterns>
</window-pattern>
```

An event streams into the notification window.

```
<window-notification name='traderBusted'>
   <smtp host='smtp-server.ec.rr.com'
```

```
      user='esptest@ec.rr.com'
      password='esptest1' port='587' />
<schema>
   <fields>
      <field name='id' type='int64' key='true' />
      <field name='broker' type='int32' />
      <field name='brokerName' type='string' />
      <field name='email' type='string' />
      <field name='phone' type='string' />
      <field name='sms' type='string' />
      <field name='mms' type='string' />
      <field name='customer' type='int32' />
      <field name='symbol' type='string' />
      <field name='tstamp1' type='string' />
      <field name='tstamp2' type='string' />
      <field name='tstamp3' type='string' />
      <field name='tradeId1' type='int32' />
      <field name='tradeId2' type='int32' />
      <field name='tradeId3' type='int32' />
      <field name='price1' type='double' />
      <field name='price2' type='double' />
      <field name='price3' type='double' />
      <field name='quant1' type='int32' />
      <field name='quant2' type='int32' />
      <field name='quant3' type='int32' />
      <field name='slot' type='int32' />
      <field name='day' type='string' />
      <field name='price1' type='double' />
      <field name='price2' type='double' />
      <field name='price3' type='double' />
      <field name='time1' type='string' />
      <field name='time2' type='string' />
      <field name='time3' type='string' />
      <field name='profit' type='double' />
   </fields>
</schema>
<function-context>
   <properties>
      <property-list name='time1' delimiter=' '>$tstamp1</property-list>
      <property-list name='time2' delimiter=' '>$tstamp2</property-list>
      <property-list name='time3' delimiter=' '>$tstamp3</property-list>
   </properties>
   <functions>
      <function name='profit'>product($quant3,diff($price3,$price1))</function>
      <function name='day'>listItem(#time1,0)</function>
      <function name='time1'>listItem(#time1,1)</function>
      <function name='time2'>listItem(#time2,1)</function>
      <function name='time3'>listItem(#time3,1)</function>
      <function name='price1'>precision($price1,2)</function>
      <function name='price2'>precision($price2,2)</function>
      <function name='price3'>precision($price3,2)</function>
   </functions>
</function-context>
<delivery-channels>
   <email test='true' throttle-interval='1 day'>
      <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
```

```
   <email-info>
      <sender>esptest@ec.rr.com</sender>
      <recipients>$email</recipients>
      <from>ESP Broker Surveillance</from>
      <to>$brokerName</to>
      <subject>You have been caught cheating, $brokerName</subject>
   </email-info>
   <email-contents>
      <html-content><![CDATA[
      <body>You bought <b>$quant1</b> shares of <b>$symbol</b>
            for $<b>$price1</b> on <b>$day</b> at <b>$time1</b>.
            You then bought <b>$symbol</b> for customer <b>$customer</b>
            at <b>$time2</b>, after which you sold <b>$quant3</b> shares of
            <b>$symbol</b> at <b>$time3</b> for $<b>$price3</b>,
            thus making you a profit of $<b>$profit</b>. <br/><br/></body>
      ]]></html-content>
      <image-content type='image'>
       http://esp-base:18080/esp/stuff/jail.jpg
      </image-content>
   </email-contents>
</email>
<mms test='true' throttle-interval='1 day'>
   <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
   <mms-info>
      <sender>esptest@ec.rr.com</sender>
      <subject>You have been caught cheating, $brokerName</subject>
      <gateway>$mms</gateway>
      <phone>$phone</phone>
   </mms-info>
   <mms-contents>
      <text-content>You bought $quant1 shares of $symbol
       for $$price1 on $day at $time1. You then bought $symbol for customer
       $customer at $time2, after which you sold $quant3 shares of $symbol
       at $time3 for $$price3, thus making you a profit of $$profit.
      </text-content>
      <image-content type='image'>
       http://esp-base:18080/esp/stuff/x.jpg
      </image-content>
   </mms-contents>
</mms>
<sms test='true' throttle-interval='1 day'>
   <deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
   <sms-info>
      <sender>esptest@ec.rr.com</sender>
      <subject>You have been caught, $brokerName</subject>
      <from>ESP Broker Surveillance</from>
      <gateway>$sms</gateway>
      <phone>$phone</phone>
   </sms-info>
   <sms-contents>
      <text-content>You bought $quant1 shares of $symbol
      for $$price1 on $day at $time1. You then bought $symbol
      for customer $customer at $time2, after which you sold
      $quant3 shares of $symbol at $time3 for $$price3,
      thus making you a profit of $$profit.</text-content>
   </sms-contents>
```

```
        </sms>
      </delivery-channels>
    </window-notification>
```

Because this example uses MMS, you need to define a different SMTP server. Any email account referenced by that server must be specified in your SMTP configuration. The window calculates fields to use when formatting notification messages to the broker. A schema and a function context are defined.

When an event comes in, functions are run on the input event and schema data is created. You can use values from either the input event or the schema data in the message content. For example:

```
We noticed you bought <b>$quant1</b> shares of <b>$symbol</b> for $<b>$price1</b>
on <b>$day</b> at <b>$time1</b>. You then bought <b>$symbol</b> for
customer <b>$customer</b> at <b>$time2</b>, after which
you sold <b>$quant3</b> shares of <b>$symbol</b> at <b>$time3</b>
for $<b>$price3</b>, thus making you a profit of $<b>$profit</b>.
```

Note the number of variable references, some to the schema data (`quant1`, `price1`, `price3`, ...), and some to the input data (`symbol`). Variable references are also used to resolve the routing information for the notification:

```
<recipients>$email</recipients>
<gateway>$sms</gateway>
<phone>$phone</phone>
```

A function is used to determine when to send the notification. The same deliver function is used for all channels.

```
<deliver>contains(toLower($brokerName),'@BROKER@')</deliver>
```

Whenever you see the notation `@TOKEN@` in an XML model, this means that the token is resolved when the project is loaded. These tokens can be resolved in one of three ways:

- on the command line, for example,**`dfesp_xml_server -BROKER pippin`**
- in your environment, for example, `$ export BROKER=pippin`
- in the properties for a project, for example, `<property name='BROKER'>pippin</property>`

In this case, you can specify which broker to use to send a notification.

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Pattern Windows

Overview of Pattern Windows

To create a pattern window, do the following:

- specify a list of events of interest (EOIs)
- connect EOIs into an expression that uses logical operators and optional temporal conditions

Specify EOIs by providing the following:

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

The valid logical operators for the pattern logic used by SAS Event Stream Processing are as follows:

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

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

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

when event2 happens within an hour of event1. Specify

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

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

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

Coding Pattern Windows

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

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

```
</patterns>
</window-pattern>
```

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

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

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

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

```
ID*:int32,symbol:string,price:double,buy:int32,tradeTime:date
```

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

```
// Someone buys (or sells IBM) at price > 100.00
//     followed within 5 seconds of selling (or buying) SUN at price
//       > 25.00
dfESPpatternUtils::patternNode *l,*r, *f;
l = p_01->addEvent(sw_01, "e1",
                   "((symbol==\"IBM\") and (price > 100.00)
                   and (b == buy))");
r = p_01->addEvent(sw_01, "e2",
                   "((symbol==\"SUN\") and (price > 25.000)
                   and (b == buy))");
f = p_01->fby_op(l, r, 5000000); // note 5,000,000 microseconds
     = 5 seconds
```

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

Because `b` is not a field in the incoming event, it is a free variable that is bound when an event arrives. It matches the first portion of the WHERE clause for event `l` (for example, an event for IBM with price > 100.00.) In this case, `b` is set to the value of the field `buy` in the matched event. This value of `b` is then used in evaluating the WHERE clause for subsequent events that are candidates for matching the second event of interest `r`. The added unification clause `and (b == buy)` in each event of interest ensures that the same value for the field `buy` appears in both matching events.

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

When you want to apply a temporal condition to the FBY operator, append the condition to the function inside braces. For example:

```
    fby{1 hour}(event1,event2)
fby{10 minutes}(event1,event2,event3)
```

In the first line of code, event2 happens within an hour of event1. In the second line, event3 happens within ten minutes of event2, which happens within ten minutes of event1.

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

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

```
// Someone buys or sells IBM at price > 100.00 and also
//      buys or sells IBM at a price > 102.00 within 5 seconds.
l = p_01->addEvent(sw_01, "e1",
                   "((symbol==\"IBM\") and (price > 100.00)");
r = p_01->addEvent(sw_01, "e2", "((symbol==\"IBM\") and (price
                                   > 102.00)");
f = p_01->and_op(l, r, 5000000); // note 5,000,000 microseconds
     = 5 seconds
```

Now suppose an event comes into the window where symbol is "IBM" and price is "102.1". Because this is an AND operator, no inherent sequencing is involved, and the WHERE clause is satisfied for both sides of the "and" by the single input event. Thus, the pattern becomes true, and event `l` is the same as event `r`. This is probably not what you intended. Therefore, you can make slight changes to the pattern as follows:

```
// Someone buys (or sells IBM) at price > 100.00 and <= 102.00
// and also buys or selld IBS) at a price > 102.00 within 5 seconds.
l = p_01->addEvent(sw_01, "e1",
                   "(symbol==\"IBM\") and (price > 100.00) and
                   (price <= 102.00)");
r = p_01->addEvent(sw_01, "e2", "(symbol==\"IBM\") and (price
     > 102.00)");
f = p_01->and_op(l, r, 5000000); // note 5,000,000 microseconds
     = 5 seconds
```

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

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

State Definitions for Operator Trees

Operator trees can have one of the following states:

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

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

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

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

Expression	Outcome
`not` *EOI*	becomes TRUE and FIXED or FALSE and FIXED on the application of a single event. It becomes TRUE if the event is applied it does not satisfy the event of interest, and FALSE if it does
`not` *OPT*	is a Boolean negation. This remains in the waiting state until *OPT* evaluates to TRUE or FALSE. Then it performs the logical negation. It becomes FIXED only when *OPT* becomes FIXED
`notoccur` *EOI*	becomes TRUE on application of an event that does not satisfy the *EOI*, but it is not marked FIXED. This implies that more events can be applied to it. As soon as it sees an event that matches the *EOI*, it becomes FALSE and FIXED
`notoccur` *OPT*	is not permitted
EO `or` *EO*	is an event that is always applied to all non-FIXED sub-trees. It becomes TRUE when one of its two sub-trees become TRUE. It becomes FALSE when both of the sub-trees becomes FASLE. It is FIXED when one of its sub-trees is TRUE and FIXED if both of its sub-trees are FALSE and not FIXED
EO `and` *EO*	is an event that is always applied to all non-FIXED sub-trees. It becomes TRUE when both of its two sub-trees become TRUE. It becomes FALSE when one of the sub-trees becomes FALSE. It is FIXED when one of its sub-trees is FALSE and FIXED or both of its sub-trees are TRUE and FIXED
EO `FBY` *EO*	attempts to complete the left hand side (LHS) with the minimal number of event applications before applying events to the right hand side (RHS). The apply rule is as follows: ■ If the LHS is not TRUE or FALSE, apply event to the LHS until it become TRUE or FALSE. ■ When the LHS becomes FALSE, set the followed by state to FALSE and become FIXED. ■ When the LHS becomes TRUE, apply all further events to the RHS until the RHS becomes TRUE or FALSE. If the RHS becomes FALSE, set the `FBY` state to FALSE and FIXED, if it becomes TRUE set the `FBY` state toTRUE and FIXED. This algorithm seeks the minimal length sequence of events that completes an `FBY` pattern.
`is` *EOI*	becomes TRUE on the application of an event that satisfies the EOI and FALSE otherwise. Becomes FIXED on the first application of an event.
`is` *OPT*	is not permitted

The following table describes the logic underlying a set of sample operator trees and events:

Logic	Description
`(a fby b)`	Detect a, ..., b, where ... can be any sequence.
`( a fby ( (notoccur c) and b ) )`	Detect a, ..., b: but there can be no c between a and b.

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

Restrictions on Patterns

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

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

```
// Someone buys (or sells) IBM at price > 100.00
//      followed within 5 seconds of selling (or buying)
//      SUN at price > 25.00 or someone buys (or sells)
//      SUN at price > 25.00 followed within 5 seconds
//            of selling (or buying) IBM at price > 100.00
//
dfESPpatternUtils::patternNode *l,*r, *lp, *rp, *fp;
l = p_01->addEvent(sw_01, "e1",
                   "((symbol==\"IBM\") and (price > 100.00)
                   and (b == buy))");
r = p_01->addEvent(sw_01, "e2", "((symbol==\"SUN\") and
                   (price > 25.000) and (b == buy))");
lp = p_01->fby_op(l, r, 5000000);   // note microseconds
rp = p_01->fby_op(r, l, 5000000);   // note microseconds
fp = p_01->or_op(lp, rp, 5000000);
```

  To obtain the desired result, you need four events of interest as follows:

```
dfESPpatternUtils::patternNode *l0,*r0, *l1, *r1, *lp, *rp, *fp;
l0 = p_01->addEvent(sw_01, "e1", "((symbol==\"IBM\") and
                                  (price > 100.00) and (b == buy))");
r0 = p_01->addEvent(sw_01, "e2", "((symbol==\"SUN\") and
                                  (price > 25.000) and (b == buy))");
l1 = p_01->addEvent(sw_01, "e3", "((symbol==\"IBM\") and
                                  (price > 100.00) and (b == buy))");
r1 = p_01->addEvent(sw_01, "e4", "((symbol==\"SUN\") and
                                  (price > 25.000) and (b == buy))");
lp = p_01->fby_op(l0, r0, 5000000);   // note microseconds
rp = p_01->fby_op(l1, r1, 5000000);   // note microseconds
fp = p_01->or_op(lp, rp, 5000000);
```

- Pattern windows work only on Insert events.

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

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

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

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

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

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

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

  Now consider **`e2`**. Ensure the following:

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

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

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

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

Using Stateless Pattern Windows

Pattern windows are insert-only with respect to both their input windows and the output that they produce. The output of a pattern window is a monotonically increasing integer ID that represents the number of patterns found in the pattern window. The ID is followed by an arbitrary number of non-key fields assembled from the fields of the events of interest for the pattern. Because both the input and output of a pattern window are unbounded and insert-only, they are natural candidates for stateless windows (that is, windows with index type `pi_EMPTY`). Usually, you want to have a copy window with a retention policy follow any insert-only window.

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

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

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

Enabling Pattern Compression

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

There are two ways to enable pattern compression on projects:

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

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

Enabling the Heartbeat Interval

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

To remedy this problem, tune the heartbeat interval:

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

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

Pattern Window Examples

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Procedural Windows

Overview to Procedural Windows

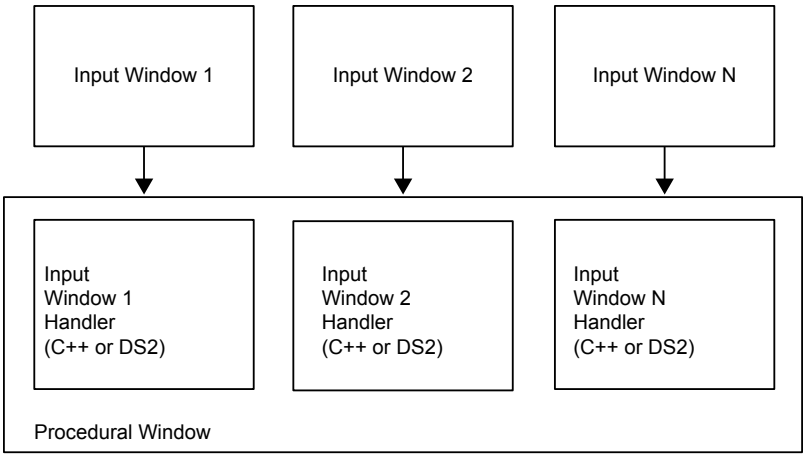
A procedural window enables you to specify an arbitrary number of input windows and input handler functions for each input window. You can define procedural window input handlers in one of the following ways:

- Use a callable C++ function in a shared library. Use the cxx-plugin XML element to define this type of input handler.
- Use DS2 code run in table server mode. An input clock is passed to a DS2 routine that produces zero or more output rows. Use the ds2–tableserver XML element to define this type of input handler.

- Use DATA step code that calls into an external SAS session that runs on the same server that produces one output row for each input row.
- Use a MAS method written in DS2 or Python and defined in a MAS (Micro Analytics Service) module. This method uses a function call interface to generate one output event for each input event.

When an input event arrives, the input handler registered for the matching window is called. Then the events produced by that input handler function are generated.

Procedural Window with Input Handlers



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

Using C++ Window Handlers

Here is an example of the signature of a procedural window handler written in C++.

```
typedef bool (*dfESPevent_func)(dfESPpcontext *pc,
              dfESPschema *is, dfESPeventPtr nep,
               dfESPeventPtr oep, dfESPschema *os,
              dfESPptrVect<dfESPeventPtr>&oe);
```

The procedural context is passed to the handler. The input schema, the new event, and the old event (in the case of an update) are passed to the handler when it is called. The final parameters are the schema of the output event (the structure of events that the procedural window produces) and a reference to a vector of output events. It is this vector where the handler needs to push its computed events.

Only one input window is defined, so define only one handler function and call it when a record arrives.

```
// This handler functions simple counts inserts, updates,
//      and deletes.
// It generates events of the form "1,#inserts,#updates,
//      #deletes"
//
bool opcodeCount(dfESPpcontext *mc, dfESPschema *is,
                    dfESPeventPtr nep, dfESPeventPtr oep,
                    dfESPschema *os, dfESPptrVect
                    <dfESPeventPtr>& oe) {

    derivedContext *ctx = (derivedContext *)mc;
    // Update the counts in the past context.
```

```
        switch (nep->getOpcode()) {
            case dfESPeventcodes::eo_INSERT:
               ctx->numInserts++;
               break;
            case dfESPeventcodes::eo_UPDATEBLOCK:
               ctx->numUpdates++;
               break;
            case dfESPeventcodes::eo_DELETE:
               ctx->numDeletes++;
               break;
          }

        // Build a vector of datavars, one per item in our output
        //      schema, which looks like: "ID*:int32,insertCount:
        //      int32,updateCount:int32,deleteCount:int32"

    dfESPptrVect<dfESPdatavarPtr> vect;
    os->buildEventDatavarVect(vect);

    // Set the fields of the record that we are going to produce.

    vect[0]->setI32(1); // We have a key of only 1, we keep updating one record.
    vect[1]->setI32(ctx->numInserts);
    vect[2]->setI32(ctx->numUpdates);
    vect[3]->setI32(ctx->numDeletes);

    // Build the output Event, and push it to the list of output
    //      events.

    dfESPeventPtr ev = new dfESPevent();
    ev->buildEvent(os, vect, dfESPeventcodes::eo_UPSERT,
                   dfESPeventcodes::ef_NORMAL);
    oe.push_back(ev);

    // Free space used in constructing output record.
    vect.free();
    return true;
```

The following example shows how this fits together in a procedural window:

```
dfESPproject *project_01;
project_01 = theEngine->newProject("project_01");

dfESPcontquery *cq_01;
cq_01 = project_01->newContquery("cq_01");

dfESPstring source_sch = dfESPstring("ID*:int32,symbol:
                                     string,price:double");
dfESPstring procedural_sch = dfESPstring("ID*:int32,insertCount:
                                         int32,updateCount:int32,
                                         deleteCount:int32");

dfESPwindow_source *sw;
sw = cq_01->newWindow_source("source window",
                             dfESPindextypes::pi_HASH,
                             source_sch);
```

```
dfESPwindow_procedural *pw;
pw = cq_01->newWindow_procedural("procedural window",
                                 dfESPindextypes::pi_RBTREE,
                                 procedural_sch);

// Create our context, and register the input window and
//      handler.
//
derivedContext *mc = new derivedContext();
mc->registerMethod(sw, opcodeCount);

pw->registerMethodContext(mc);
```

Now whenever the procedural window sees an event from the source window (sw), it calls the handler opcodeCount with the context mc, and produces an output event.

An application can use the `dfESPengine::logBadEvent()` member function from a procedural window to log events that it determines are invalid. For example, you can use the function to permit models to perform data quality checks and log events that do not pass. There are two common reasons to reject an event:

- The event contains a null value in one of the key fields.
- The opcode that is specified conflicts with the existing window index (for example, two Inserts of the same key, or a Delete of a non-existing key).

Using DS2 Window Handlers

Overview of DS2 Window Handlers

When you write a procedural window handler in the DS2 programming language, the program is declared as a character string and set in the procedural windows context.

Here is a simple example:

```
char *DS2_program_01 =
        "ds2_options cdump;"
        "data esp.out;"
        "   dcl double cost;"
        "   method run();"
        "     set esp.in;"
        "     cost = price * quant;"
        "   end;"
        "enddata;"
```

The window handler is then added to the procedural window's context, before the context is registered with the procedural window proper.

```
/* declare the next context */
dfESPpcontext *pc_01 = new dfESPpcontext;
/* register the DS2 handler in the context */
pc_01->registerMethod_DS2TS(sw_01,  DS2_program_01);
/* register the context with the procedural window */
pw_01->registerMethodContext(pc_01);
```

Note: Review C++ code that you have used with SAS Event Stream Processing 3.2. You must replace the function `registerMethod_ds2` with the function `registerMethod_DS2TS`

All fields of the input window are seen as variables in DS2 programs, so can be used in calculations. The variable _opcode is available and takes the integer values `1` (Insert), `2` (Update), `3`(Delete), `4` (Upsert), or `5` (Safe Delete). The variable _flags is available and takes the integer values `1` (Normal) or `4`(Retention). The

variables exported from the DS2 program are all the input variables plus any global variables declared. This set of variables is then filtered by the schema field names of the procedural window to form the output event.

You can retain the state of a variable in a DS2 program. The state remains valid during the life of the project. For example, the `retain` statement in the following DS2 data block makes the `sequence` variable static, maintaining state from call to call. The variable is tracked between rows within an event block and across event blocks.

```
<ds2-tableserver source='MMD1'>
    <code>
        <![CDATA[
            ds2_options cdump;
            data esp.out;
                dcl integer sequence;
                retain sequence 0;
                method run();
                    set esp.in;
                    sequence = sequence + 1;
                end;
            enddata;
        ]]>
    <\code>
 </ds2-tableserver>
```

General Structure of a DS2 Input Handler

DS2 input handlers use the following boilerplate definition:

```
ds2_options cdump;
    data esp.out;
       global_variable_declaration;   /* global variable block    */
        method run();
           set esp.in;
           computations; /* computational statements */
        end;
enddata;
```

Examples

```
input schema:
         "ID*:int32,symbol:string,size:int32,price:double"

   output (procedural schema):
          "ID*:int32,symbol:string,size:int32,price:double,cost:double"

    ds2_options cdump;
    data esp.out;
        dcl double cost;
        method run();
           set esp.in;
           cost = price * size;  /* compute the total cost */
        end;
    enddata;
```

Here is a procedural window with one input window that does no computation. It remaps the key structure, and omits some of the input fields:

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

```
   output (procedural schema):
          "kID*:int64,symbol:string,cost:double"

    ds2_options cdump;
    data esp.out;
        dcl double cost;
        dcl bigint kID;
        method run();
           set esp.in;
           kID  = 1000000000*traderID;  /* put traderID in digits 10,11, ...*/
           kID  = kID + ID;             /* put ID in digits 0,1, ... 9      */
           cost = price * size;         /* compute the total cost */
        end;
    enddata;
```

Note: This DS2 code produces the following output: {ID, symbol, size, price, traderID, cost, kID}, which when filtered through the output schema is as follows: {kID, symbol, cost}

Here is a procedural window with one input window that augments an input event with a letter grade based on a numeric grade in the input:

```
input schema:
         "studentID*:int32,testNumber*:int32,testScore:double"

   output (procedural schema):
         "studentID*:int32,testNumber*:int32,testScore:double,testGrade:string"

    ds2_options cdump;
    data esp.out;
        dcl char(1) testGrade;
        method run();
           set esp.in;
                testGrade = select
                when (testScore >= 90) 'A'
                when (testScore >= 80) 'B'
                when (testScore >= 70) 'C'
                when (testScore >= 60) 'D'
                when (testScore >= 0)  'F'
        end;
    enddata;
```

Here is a procedural window with one input window that augments an input event with the timestamp of when it was processed by the DS2 Handler:

```
input schema:
         "ID*:int32,symbol:string,size:int32,price:double"

   output (procedural schema):
          "ID*:int32,symbol:string,cost:double,processedStamp:stamp"

    ds2_options cdump;
    data esp.out;
        method run();
           set esp.in;
           processedStamp = to_timestamp(datetime());
        end;
    enddata;
```

Here is a procedural window with one input window that filters events with an even ID. It produces two identical events (with different keys) for those events with an odd ID:

```
input schema:
         "ID*:int32,symbol:string,size:int32,price:double"

   output (procedural schema):
         "ID*:int32,symbol:string,size:int32,price:double"

    ds2_options cdump;
    data esp.out;
        method run();
           set esp.in;
           if MOD(ID, 2) = 0 then return;
           output;
           ID = ID + 1;
           output;
        end;
    enddata;
```

Given this input:

```
1,ibm,1000,100.1
    2,nec,2000,29.7
    3,ibm,2000,100.7
    4,apl,1000,300.2
```

The following output is produced:

```
1,ibm,1000,100.1
    2,ibm,1000,100.1
    3,ibm,2000,100.7
    4,ibm,2000,100.7
```

Event Stream Processor to DS2 Data Type Mappings and Conversions

The following mapping of event stream processor to DS2 data types is supported:

Event Stream Processor Data Type	DS2 Data Type
ESP_INT32	TKTS_INTEGER
ESP_INT64	TKTS_BIGINT
ESP_DOUBLE	TKTS_DOUBLE
ESP_TIMESTAMP ESP DATETIME	TKTS_TIMESTAMP TKTS DATE TKTS TIME
ESP_UTF8STR	TKTS_VARCHAR TKTS CHAR

The ESP_MONEY data type is not supported.

Here is a conversion matrix. If a data type does not appear in the matrix (for example, NVarchar), conversion is not supported for it.

From/To	Integer	BigInt	Double	Date	Time	Timestamp	Char	Varchar
int32	x							
int64		x						
double			x					
datetime				x	x	x		
timestamp				x	x	x		
utf8str							x	x

DATA Step Window Handlers

Overview

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

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

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

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

Configuration

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

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

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

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

```
<ds-external source='request'
        trace='false'
        code-file='@SAS_SOURCE_DIR@/score.sas'
```

```
        connection-timeout='5'
        max-string-length='32'
/>
```

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

Referencing SAS Event Stream Processing in a DATA Step

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

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

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

Supported Data Types

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

Event Stream Processor Data Type	DATA Step Data Type
ESP_INT32	Numeric variable. ESP NULL values map to SAS missing values, and vice versa.
ESP_INT64	Numeric variable. ESP NULL values map to SAS missing values, and vice versa.
ESP_DOUBLE	Numeric variable. ESP NULL values map to SAS missing values, and vice versa.
ESP_TIMESTAMP ESP DATETIME	Numeric variable whose value is the number of seconds since Jan 1, 1960. ESP NULL values map to SAS missing values and vice versa. .
ESP_UTF8STR	Character Variable. SAS Character variables are trimmed before being returned to SAS Event Stream Processing.
ESP_MONEY	Not supported.

Known Limitations

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

MAS Modules

A MAS module, which you define at the project level, is essentially a named block of code. This block can contain one or more functions.

You define a MAS map in a procedural window to bind a function to an input window. This binding acts as the input handler for the procedural window's input window.

A procedural window can have only one MAS map. The MAS map can include one or more window maps. Each window map has four attributes:

- the module that refers to the name of a previously defined MAS module
- the module revision, which currently must be 0
- the source attribute, which is the name of the input window for which you specify the handler
- the name of the function that chooses the method defined in the MAS module

For information about the XML elements that you use to define a MAS module, see "XML Language Elements Relevant to Procedural Windows" in *SAS Event Stream Processing: XML Language Dictionary*.

Examples of Procedural Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Creating Text Category Windows

Overview to Text Category Windows

Text category windows categorize a text field in incoming events. A text category window is insert-only. The text field can generate zero or more categories, with scores.

Note: Without a SAS Contextual Analysis license, you will not have the MCO file that is required to initialize a text category window.

Example of Text Category Windows

For XML and C++ code examples of full projects, see SAS Event Stream Processing Samples and Tips .

Understanding Retention

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

For example, consider a source window with a sliding volume-based retention policy of two. That source window always contains at most two events. When an Insert arrives causing the source window to grow to three events, the event with the oldest modification time is removed. A Delete for that event is executed.

Retention Type	Description
time-based	Retention is performed as a function of the age of events. The age of an event is calculated as current time minus the last modification time of the event. Time can be driven by the system time or by a time field that is embedded in the event. A window with time-based retention uses current time set by the arrival of an event.
volume-based	Retention is based on a specified number of records. When the volume increases beyond that specification, the oldest events are removed.

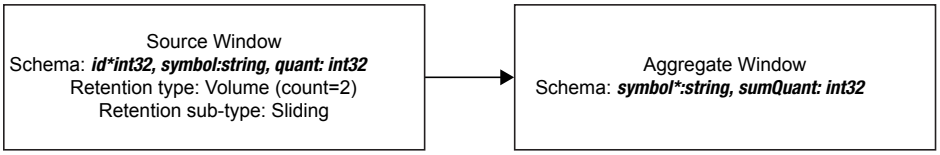
Both time and volume-based retention can occur in one of two variants:

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

A canonical set of events is a collapsed minimal set of events such that there is at most one event per key. Multiple updates for the same key and insert + multiple updates for the same key are collapsed. A window with retention generates a canonical set of changes (events). Then it appends retention-generated Deletes to the end of the canonical event set. At the end of the process, it forms the final output block.

Windows with retention produce output event blocks of the following form:`{<canonical output events>, <canonical retention deletes>}`. All other windows produce output blocks of the following form: `{<canonical output events>}`.

Consider the following model:



The following notation is used to denote events `[<opcode>/<flags>: f1, ... ,fn]`

- Opcode
  - `i` — insert
  - `d` — delete
  - `ub` — update block — any event marked as `ub` is always followed by an event marked as `d`
- Flags
  - `n` — normal
  - `r` — retention generated

Suppose that the following events are streamed into the model:

Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 1,ibm,10]`	`[i/n: 1,ibm,10]`	`[i/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 2,ibm,11]`	`[i/n: 2,ibm,11]`	`[ub/n: ibm,21]` `[d/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 3,sas,100]`	`[i/n: 3,sas,100]`	`[i/n: sas,100]`
	`[d/r: 1,ibm,10]`	`[ub/r: ibm,11]` `[d/r: ibm,21]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 4,ibm,12]`	`[i/n: 4,ibm,12]`	`[ub/r: ibm,12]` `[d/r: ibm,11]`
	`[d/r: 2,ibm,11]`	

When you run in retention-tracking mode, retention and non-retention changes are pulled through the system jointly. When the system processes a user event, the system generates a retention Delete. Both the result of the user event and the result of the retention Delete are pushed through the system. You can decide how to interpret the result. In normal retention mode, these two events can be combined to a single event by rendering its output event set canonical.

Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 1,ibm,10]`	`[i/n: 1,ibm,10]`	`[i/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out
`[i/n: 2,ibm,11]`	`[i/n: 2,ibm,11]`	`[ub/n: ibm,21]` `[d/n: ibm,10]`
Source In	Source Out — Aggregate In	Aggregate Out

[i/n: 3,sas,100]	[i/n: 3,sas,100]	[i/n: sas,100]
	[d/r: 1,ibm,10]	[ub/r: ibm,11] [d/r: ibm,21]
Source In	Source Out — Aggregate In	Aggregate Out
[i/n: 4,ibm,12]	[i/n: 4,ibm,12]	[ub: ibm,23] [d/n: ibm,11]
	[d/r: 2,ibm,11]	[ub/r: ibm,12] [d/r: ibm,23]

Here, the output of the aggregate window, because of the last input event, is non-canonical. In retention tracking mode, you can have two operations per key when the input events contain a user input for the key and a retention event for the same key.

Note: A window with pulsed mode set always generates a canonical block of output events. For the pulse to function as designed, the window buffers output events until a certain threshold time. The output block is rendered canonical before it is sent.

Understanding Primary and Specialized Indexes

Overview

In order to process events with opcodes, each window must have a primary index. That index enables the rapid retrieval, modification, or deletion of events in the window.

Windows can have other indexes that serve specialized purposes.

- source and copy windows have an additional index to aid in retention
- aggregate windows have an aggregation index to maintain the group structure
- Join windows have left and right local indexes along with optional secondary indexes. These help avoid locking and maintain data consistency.

Index Types Associated with Each Window Type

Window Type	Primary Index	Retention Index	Aggregation Index	Left Local Index	Right Local Index
Filter Window Compute Window Pattern Window Procedural Window Textcontext Window	Yes				

Window Type	Primary Index	Retention Index	Aggregation Index	Left Local Index	Right Local Index
Source Window Copy Window	Yes	Yes			
Aggregate Window	Yes		Yes		
Join Window	Yes			Yes Optional secondary	Yes Optional secondary

The `dfESPeventdepot` object that is used to store windows supports six types of primary indices: five are stateful, and one is not.

Fully Stateful Indexes

The following index types are fully stateful:

Index Type	Description
`pi_RBTREE`	Specifies red-black tree, logarithmic Insert, Deletes are O(log(n)) — provides smooth latencies. Stores events in memory.
`pi_HASH`	Specifies a typical open hash algorithm. Stores events in memory. In general this index provides faster results than `pi_RBTREE`. Unless properly sized, using this index might lead to latency spikes.
`pi_CL_HASH`	Specifies a closed hash. This index provides faster results than `pi_HASH`.
`pi_FW_HASH`	Specifies a forward hash. This index creates a smaller memory footprint than other hash indexes, but might yield poorer delete performance.
`pi_LN_HASH`	Specifies a linked hash. This index performs slightly more slowly than other hash index and uses more memory than `pi_CL_HASH`.
`pi_HLEVELDB`	On disk stateful index for large source or copy windows and for the left or right local dimension index in a join. Can be used when there is no retention, aggregation, or a need for a secondary index

For information about the closed hash, forward hash, and linked hash variants, see “Miscellaneous Container Templates” at http://www.medownloads.com/download-Miscellaneous-Container-Templates-147179.htm.

Events are absorbed, merged into the window’s index, and a canonical version of the change to the index is passed to all output windows. Any window that uses a fully stateful index has a size equal to the cardinality of the unique set of keys, unless a time or size-based retention policy is enforced.

When no retention policy is specified, a window that uses one of the fully stateful indices acts like a database table or materialized view. At any point, it contains the canonical version of the event log. Because common events are reference-counted across windows in a project, you should be careful that all retained events do not exceed physical memory.

Use the Update and Delete opcodes for published events (as is the case with capital market orders that have a lifecycle such as create, modify, and close order). However, for events that are Insert-only, you must use window retention policies to keep the event set bound below the amount of available physical memory.

Non-Stateful Index

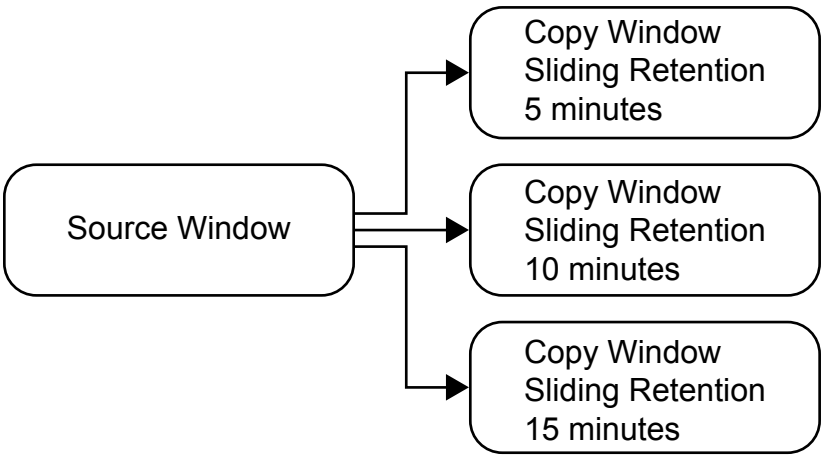
The non-stateful index is a source window that can be set to use the index type `pi_EMPTY`. It acts as a pass-through for all incoming events. This index does not store events.

The following restrictions apply to source windows that use the empty index type.

- No restrictions apply if the source window is set to "Insert only."
- If the source window is not Insert-only, then it must be followed by one of the following:
  - a copy window with a stateful index
  - a functional window or a compute window
- When a source, compute, or functional window in a linear chain with `pi_EMPTY` indexes start a model, one of the following must be true about the linear chain:
  - It must end with a functional window that converts its events to Insert only, and have the `produces-only-inserts` property set.
  - It must end in a stateful compute, functional, or copy window that convert Upserts to Inserts or have updates that can propagate further through the model automatically through the stateful index.

Using empty indices and retention enables you to specify multiple retention policies from common event streams coming in through a source window. The source window is used as an absorption point and pass-through to the copy windows, as shown in the following figure.

Copy Windows



Using the pi_HLEVELDB Primary Index with Big Dimension Tables

Overview

A common use case is to stream fact data into an engine and join it with dimension data for further processing. This works well under the following two conditions:

- The model is stateless or is controlled by retention.
- The entire dimension table fits into memory.

However, when the dimension table consists of tens or hundreds of million rows, this common use case becomes problematic. You can increase system memory to an extent, after which price and hardware limitations affect the size of the data that can be effectively processed.

With massive dimension tables, you can use the `pi_HLEVELDB` index to store events in an on-disk store. This produces large on-disk event indexes with a correspondingly small RAM footprint. Using this index type is helpful in the following circumstances:

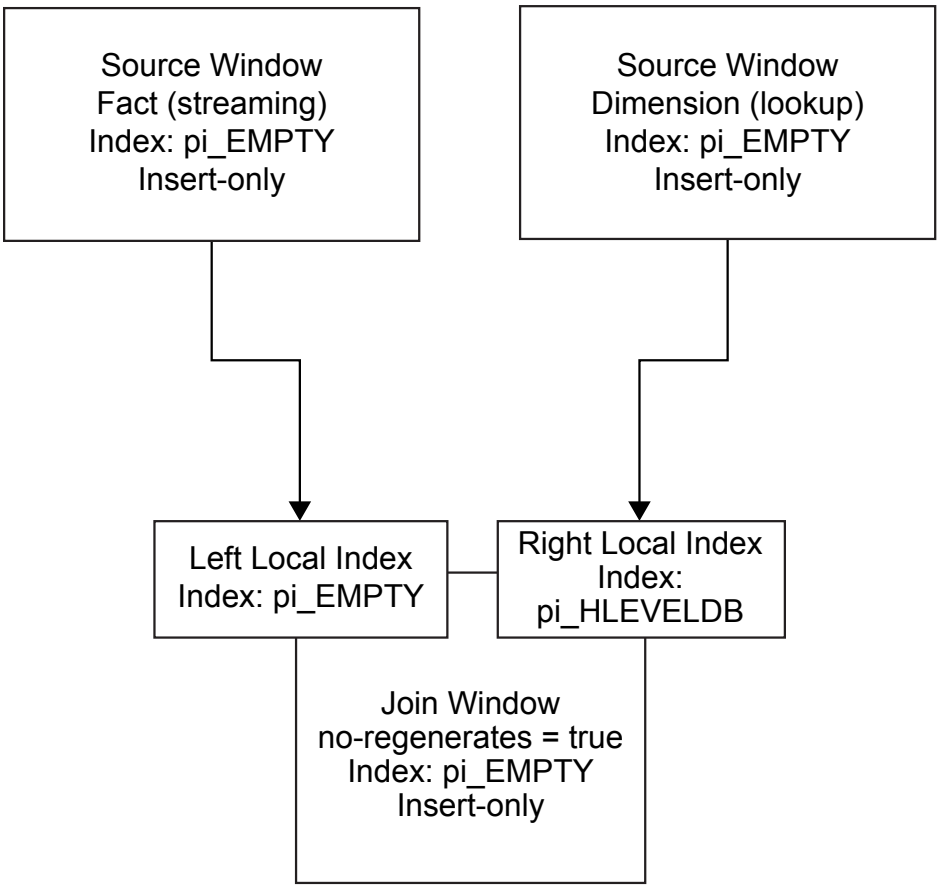
- Throughput is measured in tens of thousands of events per second or more.
- The window is not implementing retention or aggregation.
- No secondary index is used.

Stateless Left Outer Join: One-Time Bulk Load with No-Regeneration

Consider the following case. A stateless left outer join streams Insert-only data through the left window (fact) and matches it against dimensional data on the right. It passes Inserts out of the join. It uses the **`no-regenerates`** option of the join window, so future inserts to the dimension table affect only future streaming fact events.

In this model, you first prime the dimension table with a large volume of Inserts, perhaps hundreds of millions of rows. Specify that the right local index of the join have index type `pi_HLEVELDB`. This stores the dimension data in the right local index in an on-disk store. After the dimension data has been fully loaded, the fact stream can be started. Join matches are made against the dimensional events in the on-disk store, using an MRU memory cache for lookups and a filter to minimize disk seeks.

Streaming Data into a Join Window Using Two Different Stateful Index Types



The following C++ code fragment implements the model.

```
dfESPproject  *project_01 = theEngine->newProject("project_01");
        project_01->setPubSub(dfESPproject::ps_MANUAL);

    dfESPcontquery *cq_01 =    project_01->newContquery("cq_01");
         dfESPstring schema_01 =
           dfESPstring("S_ID*:string,S_Plan:string,S_gid:string,S_flag:string");

    dfESPwindow_source *Dim =
           cq_01->newWindow_source((dfESPstring)"Dim",
                  dfESPindextypes::pi_EMPTY, schema_01);
           Dim->setInsertOnly();

    dfESPwindow_source *Fact = cq_01->newWindow_source((dfESPstring)"Fact",
                           dfESPindextypes::pi_EMPTY, schema_01);
        Fact->setInsertOnly();

    dfESPwindow_join *Join =
           cq_01->newWindow_join((dfESPstring)"Join", dfESPwindow_join::jt_LEFTOUTER,
                              dfESPindextypes::pi_EMPTY, false, true);
        Join->setRightIndexType(dfESPindextypes::pi_HLEVELDB);

    Join->setJoinConditions("l_S_ID==r_S_ID");

    Join->addNonKeyFieldCalc("r_S_ID");
    Join->addNonKeyFieldCalc("l_S_Plan");
    Join->addNonKeyFieldCalc("r_S_Plan");

    Join->setFieldSignatures("r_S_ID:string,l_S_Plan:string,r_S_Plan:string");

    cq_01->addEdge(Fact, Join);
    cq_01->addEdge(Dim, Join);
```

Stateless Left Outer Join: Dimensional Updates with No-Regeneration

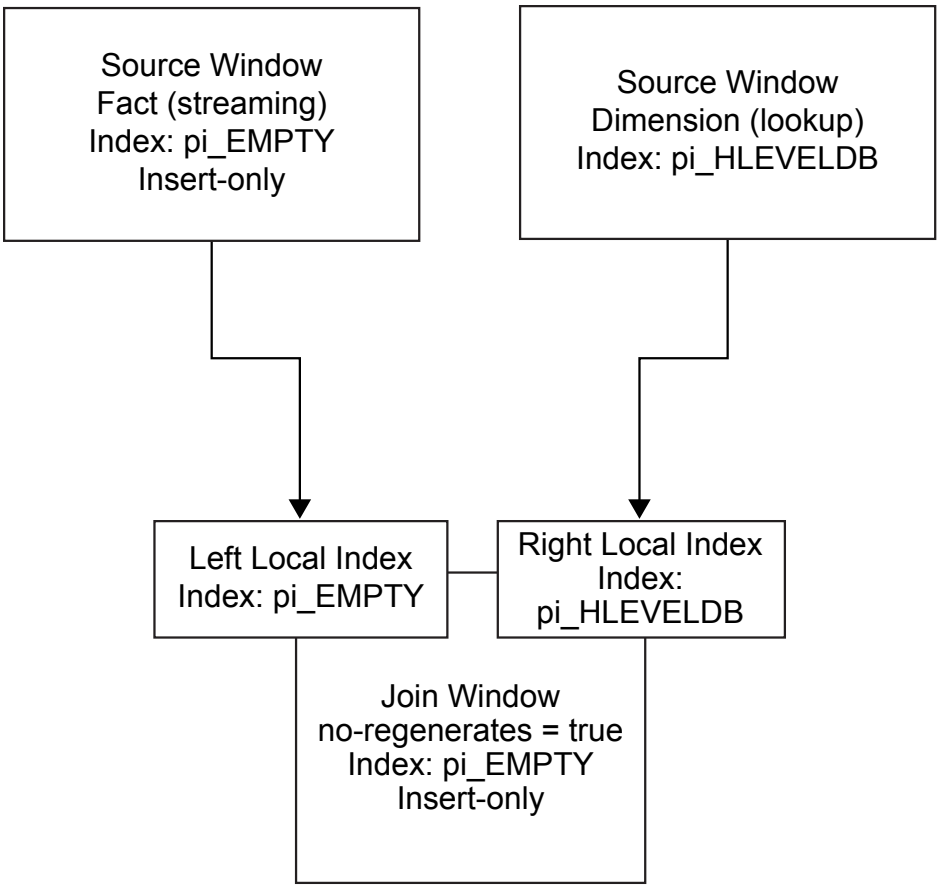
Now suppose you want to periodically update or reload the dimension table in this model. Thus, the dimensional data is no longer Insert only. To correctly resolve opcodes, the source window into which the dimensional data flows must have a stateful index.

To do this, you modify the previous model to put the dimension source data in an on-disk store. Store the following indexes to disk:

1. the primary index for the dimension source window
2. the right local index for the join

Again, the join is set to **`no-regenerates`** so that dimensional changes affect only new data. Because the `pi_HLEVELDB` index type does not support a retention policy, the dimension data should be naturally bounded. That is, it can encompass a very large number of events, but not an infinite number.

Streaming Data into a Join Window When the Dimension Table Uses the pi_HLEVELDB Index Type



The following XML code implements the model:

```
<project name='pr_01' pubsub='manual' threads='4' disk-store-path='/tmp/jones>
      <contqueries>
        <contquery name='cq_01'>
          <windows>
            <window-source name='Dim' index='pi_HLEVELDB'>
              <schema>
                <fields>
                  <field name='S_ID' type='string' key='true' />
                  <field name='S_Plan' type='string' />
                  <field name='S_gid' type='string' />
                  <field name='S_flag' type='string' />
                </fields>
              </schema>
            </window-source>
            <window-source name='Fact' index='pi_EMPTY' insert-only='true'>
              <schema>
                <fields>
                  <field name='S_ID' type='string' key='true' />
                  <field name='S_Plan' type='string' />
                  <field name='S_gid' type='string' />
                  <field name='S_flag' type='string' />
                </fields>
              </schema>
            </window-source>
```

```
        <window-join name='join_w' index='pi_EMPTY'>
          <join type='leftouter' left='Fact' right='Dim'
                right-index='pi_HLEVELDB' no-regenerates='true'>
            <conditions>
              <fields left='S_ID' right='S_ID'/>
            </conditions>
          </join>
          <output>
            <field-selection name='r_S_ID' source='r_S_ID'/>
            <field-selection name='l_S_Plan' source='l_S_Plan'/>
            <field-selection name='r_S_Plan' source='r_S_Plan'/>
          </output>
        </window-join>
      </windows>
      <edges>
        <edge source='Fact' target='join_w'/>
        <edge source='Dim' target='join_w'/>
      </edges>
    </contquery>
  </contqueries>
</project>
```

Design Patterns

Overview to Design Patterns

A design pattern is a reusable solution to a common problem within a specific context of software design. The combinations of windows that you use in your design pattern should enable fast and efficient event stream processing.

Event stream processing models can be stateless, stateful, or mixed. The type of model that you choose affects how you design it. One challenge when you design a mixed model is to identify sections that must be stateful and those that can be stateless, and then connecting them properly.

A stateless model is one where the indexes on all windows have the type **`pi_EMPTY`**. Events are not retained in any window, and are essentially transformed and passed through. Stateless models exhibit fast performance and use very little memory. They are well-suited to tasks where the inputs are inserts and when simple filtering, computation, text context analysis, or pattern matching are the only operations you require.

A stateful model is one that uses windows with index types that store data, usually **`pi_RBTREE`** or **`pi_HASH`**. These models can fully process events with Insert, Update, or Delete opcodes. A stateful model facilitates complex relational operations such as joins and aggregations. Because events are retained in indexes, whenever all events are Inserts only, windows grow unbounded in memory. Thus, stateful models must process a mix of Inserts, Updates, and Deletes in order to remain bounded in memory.

The mix of opcodes can occur in one of two ways:

- The data source and input events have bounded key cardinality. That is, there are a fixed number of unique keys (such as customer IDs) in the input stream. You can make many updates to these keys provided that the key cardinality is finite.
- A retention policy is enforced for the data flowing in, where the amount of data is limited by time or event count. The data is then automatically deleted from the system by the generation of internal retention delete events.

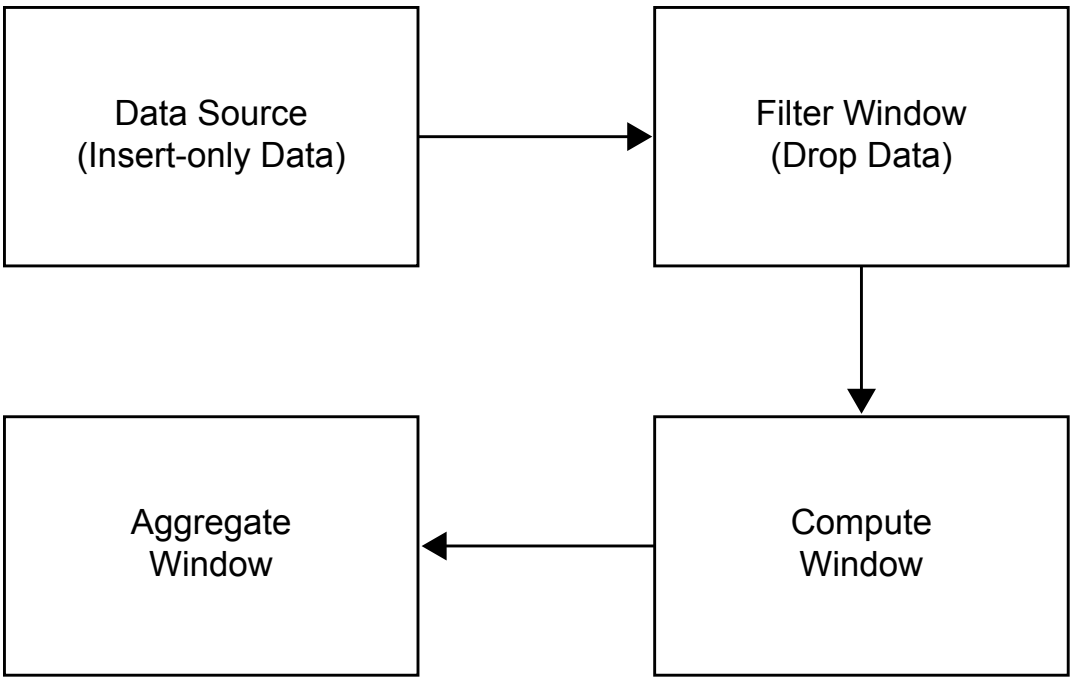
A mixed model has stateless and stateful parts. Often it is possible to separate the parts into a stateless front end and a stateful back end.

Design Pattern That Links a Stateless Model with a Stateful Model

To control memory growth in a mixed model, link the stateless and stateful parts with copy windows that enforce retention policies. Use this design pattern when you have insert-only data that can be pre-processed in a stateless way. Pre-process the data before you flow it into a section of the model that requires stateful processing (using joins, aggregations, or both).

For example, consider the following model:

Event Stream Processing Model with Insert-Only Data



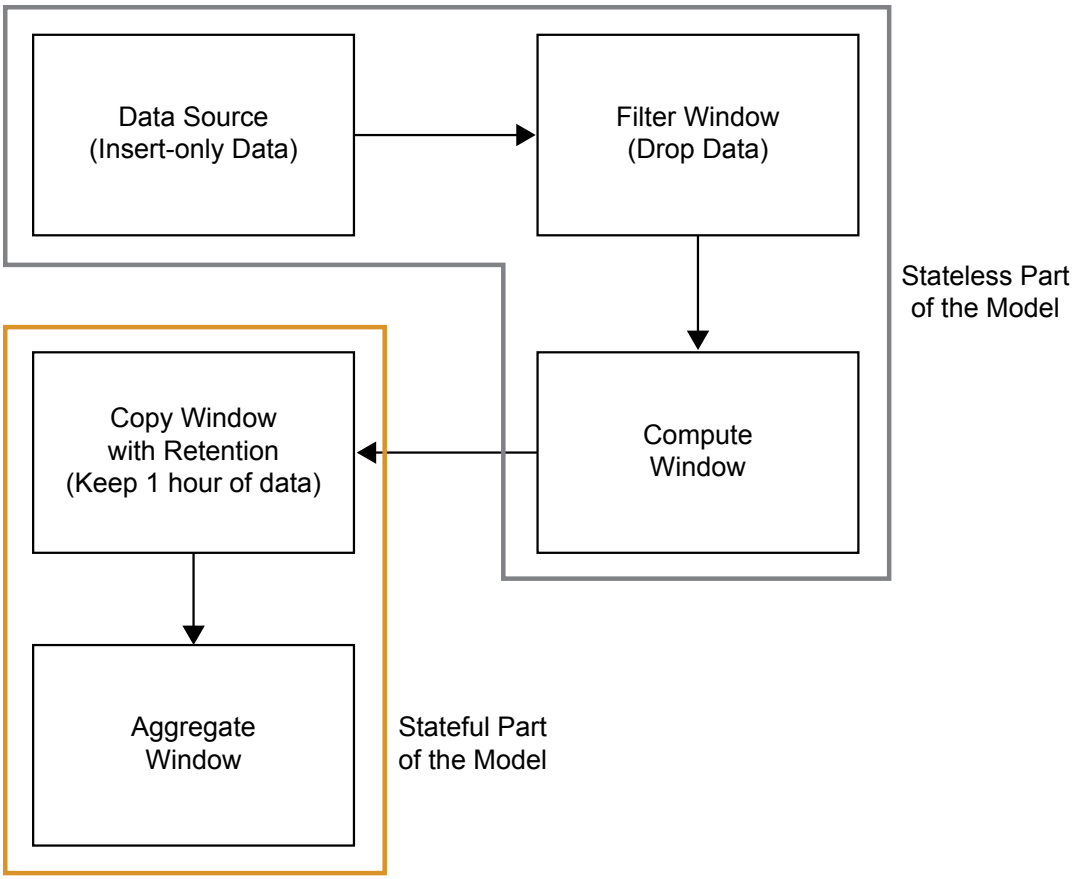
Here the data source is purely through Inserts. Therefore, the model can be made stateless by using an index type of **`pi_EMPTY`**. The filter receives inserts from the source, and drops some of them based on the filter criteria, so it produces a set of inserts as output. Thus, the filter can be made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

The compute window transforms the incoming inserts by selecting some of the output fields of the input events. The same window computes other fields based on values of the input event. It generates only inserts, so it can be stateless.

After the compute window, there is an aggregate window. This window type needs to retain events. Aggregate windows group data and compress groups into single events. If an aggregate window is fed a stream of Inserts, it would grow in an unbounded way.

To control this growth, you can connect the two sections of the model with a copy window with a retention policy.

Modified Event Stream Processing Model with Stateless and Stateful Parts



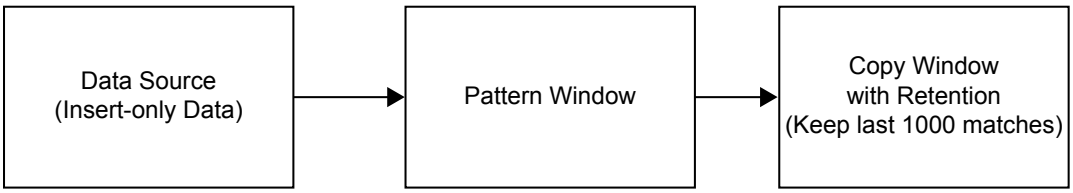
The stateful part of the model is accurately computed, based on the rolling window of input data. This model is bounded in memory.

Controlling Pattern Window Matches

Pattern matches that are generated by pattern windows are Inserts. Suppose you have a source window feeding a pattern window. Because a pattern window generate Inserts only, you should make it stateless by specifying an index type of `pi_EMPTY`. This prevents the pattern window from growing infinitely. Normally, you want to keep some of the more recent pattern matches around. Because you do not know how frequent the pattern generates matches, follow the pattern window with a count-based copy window.

Suppose you specify to retain the last 1000 pattern matches in the copy window.<

Event Stream Processing Model with Copy with Retention



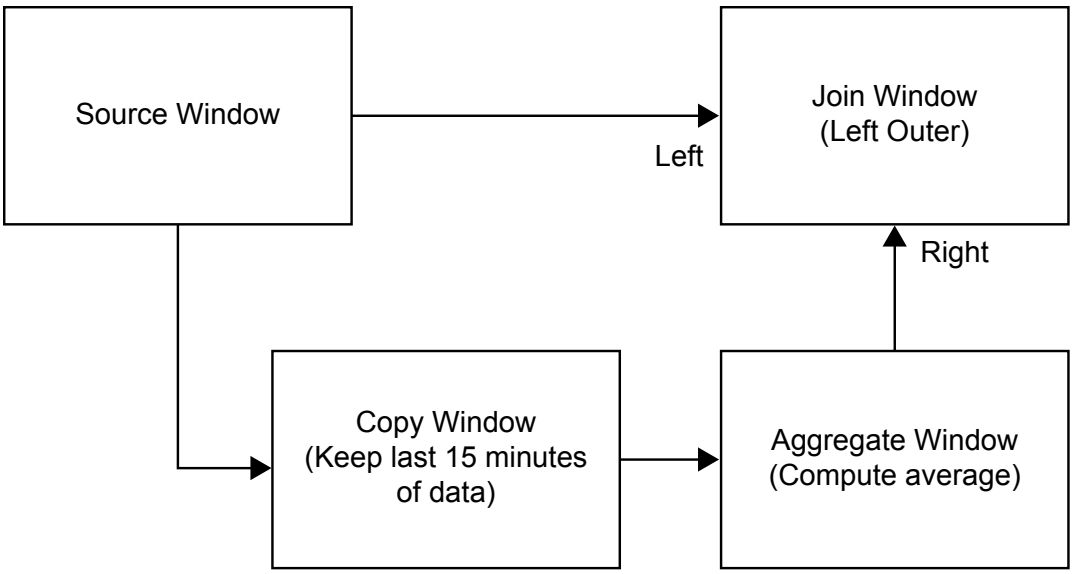
In cases like these, it is more likely that the copy window is queried from the outside using adapters, or publish/subcribe clients. The copy window might also feed other sections of the model.

Augmenting Incoming Events with Rolling Statistics

Suppose you have an insert stream of events, and one or more values are associated with the events. You want to augment each input event with some rolling statistics and then produce the augmented events. Solving this problem requires using advanced features of the modeling environment.

For example, suppose you have a stream or stock trades coming in and you want to augment them with the average stock price in the past. You build the following model.

Event Stream Processing Model Using Advanced Features



To control the aggregate window:

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

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

To process input in this way:

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

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

The following XML code implements this model.

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

                    <windows>
                        <window-source name='Trades'
                                       index='pi_RBTREE'>
                            <schema>
                                <fields>
                                    <field name='tradeID' type='string' key='true'/>
                                    <field name='security' type='string'/>
                                    <field name='quantity' type='int32'/>
                                    <field name='price' type='double'/>
                                    <field name='traderID' type='int64'/>
                                    <field name='time' type='stamp'/>
                                </fields>
                            </schema>
                        </window-source>

                        <window-copy name='TotalIn'>
                            <retention type='bycount_sliding'>5</retention>
                        </window-copy>

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

                        <window-join name='JoinTotal'
                                     index='pi_EMPTY'>
                            <join type="leftouter"
                                  no-regenerates='true'>
                                <conditions>
                                  <fields left='tradeID' right='tradeID'/>
                                  <fields left='security' right='security'/>
                                </conditions>
                            </join>
                            <output>
                                <field-selection name='quantity'
                                  source='l_quantity'/>
```

```
                        <field-selection name='price'
                          source='l_price'/>
                        <field-selection name='traderID'
                          source='l_traderID'/>
                        <field-selection name='time'
                          source='l_time'/>
                        <field-selection name='quantityTotal'
                          source='r_quantityTotal'/>
                    </output>
                </window-join>
            </windows>

            <edges>
                <edge source='Trades'
                      target='JoinTotal'/>
                <edge source='Trades'
                      target='TotalIn'/>
                <edge source='TotalIn'
                      target='RunTotal'/>
                <edge source='RunTotal'
                      target='JoinTotal'/>
            </edges>

        </contquery>
    </contqueries>
</project>
</projects>
</engine>
```

Advanced Window Operations

Implementing Periodic (or Pulsed) Window Output

In most cases, the SAS Event Stream Processing API is fully event driven. That is, windows continuously produce output as soon as they transform input. But there might be times when you want a window to hold data and then output a canonical batch of updates. In this case, operations to common key values are collapsed into a single operation.

Here are two cases where batched output might be useful:

- Visualization clients might want to get updates once a second because they cannot visualize changes any faster than this. When the event data is pulsed, the clients take advantage of the reduction of event data to visualize through the collapse around common key values.
- A window that follows the pulsed window is interested in comparing the deltas between periodic snapshots from that window.

Use the following call to add output pulsing to a window:

```
dfESPwindow::setPulseInterval(size_t us);
```

Note: Periodicity is specified in microseconds. However, given the clock resolution of most non-real-time operating systems, the minimum value that you should specify for a pulse period is 100 milliseconds.

Splitting Generated Events across Output Slots

Overview

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

Most windows send all generated events out of output slot 0 to zero of more downstream windows. For this reason, it is not standard for most models to use splitters. Using window splitters can be more efficient than using filter windows off a single output slot. This is especially true, for example, when you are performing an alpha-split across a set of trades or a similar task.

Using window splitters is more efficient than using two or more subsequent filter windows. This is because the filtering is performed a single time at the window splitter rather than multiple times for each filter. This results in less data movement and processing.

Splitter Functions

Here is a prototype for a splitter function.

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

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

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

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

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

When adding an edge between windows in a continuous query, specify the slot number of the parent window where the receiving window receives its input events. If the slot number is -1, it receives all the data produced by the parent window regardless of the splitter function.

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

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

Splitter Expressions

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

```
sw_01->setSplitter("splitField%2");
```

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

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

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

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

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

The full syntax for this call is as follows:

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

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

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

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

Marking Events as Partial-Update on Publish

Overview

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

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

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

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

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

Publishing Partial Events into a Source Window

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

- In order to construct the partial event, you must represent all the fields in the event. Specify either the field type and value or a placeholder field that indicates that the field value and type are missing. In this way, the existing field value for this key field combination remains for the updated event. These field values and types can be provided as `datavars` to build the event. Alternatively, they can be provided as a comma-separated value (CSV) string.

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

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

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

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

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

Examples

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

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

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

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

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

```
dfESPeventcodes::ef_PARTIALUPDATE);
```

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

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

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

Persist and Restore Operations

SAS Event Stream Processing enables you to do the following:

- persist a complete model state to a file system
- restore a model from a persist directory that had been created by a previous persist operation
- persist and restore an entire engine
- persist and restore a project

To create a persist object for a model, provide a pathname to the class constructor: `dfESPpersist(char *baseDir);` The *baseDir* parameter can point to any valid directory, including disks shared among multiple running event stream processors.

After providing a pathname, call either of these two public methods:

```
bool persist();
bool restore(bool dumpOnly=false);
// dumpOnly = true means do not restore, just walk and print info
```

The `persist()` method can be called at any time. Be aware that it is expensive. Event block injection for all projects is suspended, all projects are quiesced, persist data is gathered and written to disk, and all projects are restored to normal running state.

The `restore()` method should be invoked only before any projects have been started. If the persist directory contains no persist data, the `restore()` call does nothing.

The persist operation is also supported by the C, Java, and Python publish/subscribe APIs. These API functions require a *host*:*port* parameter to indicate the target engine.

The C publish/subscribe API method is as follows: `int C_dfESPpubsubPersistModel(char *hostportURL, const char *persistPath)`

The Java publish/subscribe API method is as follows: `boolean persistModel(String hostportURL, String persistPath)`

One application of the persist and restore feature is saving state across event stream processor system maintenance. In this case, the model includes a call to the `restore()` function described previously before starting any projects. To perform maintenance at a later time on the running engine:

1. Pause all publish clients in a coordinated fashion.
2. Make one client execute the publish/subscribe persist API call described previously.
3. Bring the system down, perform maintenance, and bring the system back up.
4. Restart the event stream processor model, which executes the `restore()` function and restores all windows to the states that were persisted in step 2.
5. Resume any publishing clients that were paused in step 1.

To persist an entire engine, use the following functions:

```
bool dfESPengine::persist(const char * path);

void dfESPengine::set_restorePath(const char *path);
```

The path that you specify for `persist` can be the same as the path that you specify for `set_restorePath`.

To persist a project, use the following functions:

```
bool dfESPproject::persist(const char *path)

bool dfESPproject::restore(const char *path);
```

Start an engine and publish data into it before you persist it. It can be active and receiving data when you persist it.

To persist an engine, `call dfESPengine::persist(path);`. The system does the following:

1. pauses all incoming messages (suspends publish/subscribe)
2. finish processing any queued data
3. after all queued data has been processed, persist the engine state to the specified directory, creating the directory if required
4. after the engine state is persisted, resume publish/subscribe and enable new data to flow into the engine

To restore an engine, initialize it and call `dfESPengine::set_restorePath(path);`. After the call to `dfESPengine::startProjects()` is made, the entire engine state is restored.

To persist a project call `dfESPproject::persist(path);`. The call turns off publish/subscribe, quiesces the system, persists the project, and then re-enables publish/subscribe. The path specified for restore is usually the same as that for persist.

To restore the project, call `dfESPproject::restore(path);` before the project is started. Then call `dfESPengine::startProject(project);`

Gathering and Saving Latency Measurements

The `dfESPlatencyController` class supports gathering and saving latency measurements on an event stream processing model. Latencies are calculated by storing 64-bit microsecond granularity timestamps inside events that flow through windows enabled for latency measurements.

In addition, latency statistics are calculated over fixed-size aggregations of latency measurements. These measurements include average, minimum, maximum, and standard deviation. The aggregation size is a configurable parameter. You can use an instance of the latency controller to measure latencies between any source window and some downstream window that an injected event flows through.

The latency controller enables you to specify an input file of event blocks and the rate at which those events are injected into the source window. It buffers the complete input file in memory before injecting to ensure that disk reads do not skew the requested inject rate.

Specify an output text file that contains the measurement data. Each line of this text file contains statistics that pertain to latencies gathered over a bucket of events. The number of events in the bucket is the configured aggregation size. Lines contain statistics for the next bucket of events to flow through the model, and so on.

Each line of the output text file consists of three tab-separated columns. From left to right, these columns contain the following:

- the maximum latency in the bucket
- the minimum latency in the bucket

- the average latency in the bucket

You can configure the aggregation size to any value less than the total number of events. A workable value is something large enough to get meaningful averages, yet small enough to get several samples at different times during the run.

If publish/subscribe clients are involved, you can also modify publisher/subscriber code or use the file/socket adapter to include network latencies as well.

To measure latencies inside the model only:

1. Include "`int/dfESPlatencyController.h`" in your model, and add an instance of the `dfESPlatencyController` object to your `main()`.
2. Call the following methods on your `dfESPlatencyController` object to configure it:

Method	Description
`void set_playbackRate(int32_t r)`	Sets the requested inject rate.
`void set_bucketSize(int32_t bs)`	Sets the bucketSize parameter previously described.
`void set_maxEvents(int32_t me)`	Sets the maximum number of events to inject.
`void set_oFile(char *ofile)`	Sets the name of the output file containing latency statistics.
`void set_iFile(char *ifile)`	Sets the name of the input file containing binary event block data.
`void set_stampBlkSize(int64_t stampsize)`	Specifies the block size (in number of events) of memory to allocate for storing timestamps when in latency mode. Additional blocks are allocated as required.
`void set_skipSize(int32_t ss)`	Specifies the number of beginning and ending aggregation blocks to ignore in latency calculations.

3. Add a subscriber callback to the window where you would like the events to be timestamped with an ending timestamp. Inside the callback add a call to this method on your `dfESPlatencyController` object: `void record_output_events(dfESPeventblock *ob)`. This adds the ending timestamp to all events in the event block.
4. After starting projects, call these methods on your `dfESPlatencyController` object:

Method	Description
`void set_injectPoint(dfESPwindow_source *s)`	Sets the source window in which you want events time stamped with a beginning timestamp.
`void read_and_buffer()`	Reads the input event blocks from the configured input file and buffers them.

Method	Description
`void playback_at_rate()`	Time stamps input events and injects them into the model at the configured rate, up to the configured number of events.

5 Quiesce the model and call this method on your `dfESPlatencyController`object: `void generate_stats()`. This writes the latency statistics to the configured output file.

To measure model and network latencies by modifying your publish/subscribe clients:

1 In the model, call the `dfESPengine setLatencyMode()` function before starting any projects.

2 In your publisher client application, immediately before calling `C_dfESPpublisherInject()`, call `C_dfESPlibrary_getMicroTS()` to get a current timestamp. Loop through all events in the event block and for each one call `C_dfESPevent_setMeta(event, 0, timestamp)` to write the timestamp to the event. This records the publish/subscribe inject timestamp to meta location 0.

3 The model inject and subscriber callback timestamps are recorded to meta locations 2 and 3 in all events automatically because latency mode is enabled in the engine.

4 Add code to the inject loop to implement a fixed inject rate. See the latency publish/subscribe client example for sample rate limiting code.

5 In your subscriber client application, include "`int/dfESPlatencyController.h`" and add an instance of the `dfESPlatencyController` object.

6 Configure the latency controller bucketSize and playbackRate parameters as described previously.

7 Pass your latency controller object as the context to `C_dfESPsubscriberStart()` so that your subscriber callback has access to the latency controller.

8 Make the subscriber callback pass the latency controller to `C_dfESPlatencyController_recordOutputEvents()`, along with the event block. This records the publish/subscribe callback timestamp to meta location 4.

9 When the subscriber client application has received all events, you can generate statistics for latencies between any pair of the four timestamps recorded in each event. First call `C_dfESPlatencyController_setOFile()` to set the output file. Then write the statistics to the file by calling `C_dfESPlatencyController_generateStats()` and passing the latency controller and the two timestamps of interest. The list of possible timestamp pairs and their time spans are as follows:

   - (0, 2) – from inject by the publisher client to inject by the model
   - (0, 3) – from inject by the publisher client to subscriber callback by the model
   - (0, 4) – from inject by the publisher client to callback by the subscriber client (full path)
   - (2, 3) – from inject by the model to subscriber callback by the model
   - (2, 4) – from inject by the model to callback by the subscriber client
   - (3, 4) – from subscriber callback by the model to callback by the subscriber client

10 To generate further statistics for other pairs of timestamps, reset the output file and call `C_dfESPlatencyController_generateStats()` again.

To measure model and network latencies by using the file/socket adapter, run the publisher and subscriber adapters as normal but with these additional switches:

Publisher	
—r *rate*	Specifies the requested transmit rate in events per second.
-m *maxevents*	Specifies the maximum number of events to publish.
-p	Specifies to buffer all events prior to publishing.
-n	Enables latency mode.

Subscriber	
-r *rate*	Specifies the requested transmit rate in events per second.
-a *aggrsize* <*aggr_blocks_to_ignore*>	Specifies the aggregation bucket size. Can be followed by a comma-separated value that specifies the beginning and ending aggregation blocks to ignore in latency calculations.
-n	Enables latency mode.
-N *latencyblksize*	Specifies the block size (in number of events) of memory to allocate for storing timestamps when in latency mode. Additional blocks are allocated as required.

The subscriber adapter gathers all four timestamps described earlier for the windows specified in the respective publisher and subscriber adapter URLs. At the end of the run, it writes the statistics data to files in the current directory. These files are named "latency_**`transmit rate_high timestamp_low timestamp`**", where the high and low timestamps correspond to the timestamp pairs listed earlier.

Enabling Finalized Callback

Some data structures are fully created when windows and edges are made, but are finalized just before the project is started. These data structures include derived schema and certain types of window indexes. The finalized callback function is called when all data structures are completely initialized, but before any events start to flow into the window. The finalized callback function can initialize some state or connection information that is required by an application or XML model.

Enable finalized callback as follows:

- Use the following function in C++: `dfESPwindow::addFinalizeCallback(dfESPwindowCB_func cbf)`
- Use the `finalized-callback` element in XML. Specify the name of the library that contains the window callback function and the name of the function that the window calls.

```
<finalized-callback name='library' function='fin_callback'>
```

For an example, see **`$DFESP_HOME/examples/xml/procedural_kmeans`**.