SYBASE®
An SAP® Company

SPLASH Programmers Reference

Sybase Event Stream Processor 5.0

DOCUMENT ID: DC01621-01-0500-02
LAST REVISED: December 2011

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

CHAPTER 1 SPLASH Programming Language

This chapter describes the Streaming Platform LAnguage SHell (SPLASH) programming language used within Flex operators and global and local declare blocks.

The syntax of SPLASH is a combination of the expression language and a C-like syntax for blocks of statements. Just as in C, there are variable declarations within blocks, and statements for making assignments to variables, conditionals and looping. Other datatypes, beyond scalar types, are also available within SPLASH, including types for records, collections of records, and iterators over those records. Comments can appear as blocks of text inside `/*-*/` pairs, or as line comments with `//`.

Variable and Type Declarations

SPLASH variable declarations resemble those in C: the type precedes the variable names, and the declaration ends in a semicolon. The variable can be assigned an initial value as well.

Here are some examples of SPLASH declarations:

```
integer a, r;
float b := 9.9;
string c, d := 'dd';
[ integer key1; string key2; | string data; ] record;
```

The first three declarations are for scalar variables of types `integer`, `float`, and `string`. The first has two variables. In the second, the variable "b" is initialized to 9.9. In the third, the variable "c" is not initialized but "d" is. The fourth declaration is for a record with three columns. The key columns "key1" and "key2" are listed first before the `|` character; the remaining column "data" is a non-key column. The syntax for constructing new records is parallel to this syntax type.

The `typeof` operator provides a convenient way to declare variables. For instance, if `rec1` is an expression with type `[ integer key1; string key2; | string data; ]`.

```
typeof(rec1) rec2;
```

The above declaration is the same as the following declaration:

```
[ integer key1; string key2; | string data; ] rec2;
```

SPLASH type declarations also resemble those in C. The `typedef` operator provides a way to define a synonym for a type expression.

```
typedef float newFloatType;
typedef [ integer key1; string key2; | string dataField; ] rec_t;
```

These declarations create new synonyms `newFloatType` and `rec_t` for the float type and the given record type, respectively. Those names can then be used in subsequent variable declarations which improves the readability and the size of the declarations:

```
newFloatType var1;
rec_t var2;
```

Functions

You can write your own functions in SPLASH. They can be declared in global blocks, for use by any stream, or local blocks. A function can internally call other functions, or call themselves recursively.

The syntax of SPLASH functions resembles C. In general, a function looks like:

```
type functionName(type1 arg1, ..., typen argn) { ... }
```

Each “fuction type” is a SPLASH type, and each `arg` is the name of an argument. Within the `{...}` can appear any SPLASH statements. The value returned by the function is the value returned by the `return` statement within.

Here are some examples:

```
int32 factorial(integer x) {
   if (x <= 0) {
     return 1;
   } else {
     return factorial(x-1) * x;
   }
 }
string odd(integer x) {
   if (x = 1) {
      return 'odd';
   } else {
      return even(x-1);
   }
 }

string even(integer x) {
   if (x = 0) {
     return 'even';
   } else {
    return odd(x-1);
   }
}
int32 sum(integer x, integer y) { return x+y; }
string getField([ integer k; | string data;] rec) { return rec.data;}
```

The first function is recursive. The second and third are mutually recursive; unlike C, you do not need a prototype of the “even” function in order to declare the “odd” function. The last two functions illustrate multiple arguments and record input.

The real use of SPLASH functions is to define, and debug, a computation once. Suppose, for instance, you have a way to compute the value of a bond based on its current price, its days to maturity, and forward projections of inflation. You might write this function and use it in many places within the project:

```
float bondValue(float currentPrice,
                integer daysToMature,
                float inflation)
{
  ...
}
```

Integrating SPLASH into CCL with Flex Operators

Procedures written in SPLASH are integrated into Projects using the CCL Flex operator.

Procedures written in SPLASH are not meant to be standalone programs. They are meant to be used in Sybase® Event Stream Processor projects that are primarily written in CCL. The following examples show complete projects that incorporate SPLASH code using the CCL Flex operator.

This project displays the top three prices for each stock symbol.

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

/* ****************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ****************************************************
 * Use Case a:
 *         Keep records corresponding to only the top three
 * distinct values. Delete records that falls of the top
 * three values.
```

```
 *
 * Here the trades corresponding to the top three prices
 * per Symbol is maintained. It uses
 * - eventcaches
 * - local UDF
 */
CREATE FLEX Top3TradesFlex
    IN QTrades
    OUT OUTPUT WINDOW Top3Trades SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
        DECLARE
            eventCache(QTrades[Symbol], manual, Price asc)
tradesCache;
            /*
             * Inserts record into cache if in top 3 prices and
returns
             * the record to delete or just the current record if it
was
             * inserted into cache with no corresponding delete.
             */
            typeof(QTrades) insertIntoCache( typeof(QTrades)
qTrades )
            {
               // keep only the top 3 distinct prices per symbol in
the
                // event cache
                integer counter := 0;
                typeof (QTrades) rec;
                long cacheSz := cacheSize(tradesCache);
                while (counter < cacheSz) {
                    rec := getCache( tradesCache, counter );
                 if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                        // if the price is the same update
                        // the record.
                        deleteCache(tradesCache, counter);
                        insertCache( tradesCache, qTrades );
                        return rec;
                        break;
                    } else if( qTrades.Price < rec.Price) {
                        break;
                    }
                    counter++;
                }

                //Less than 3 distinct prices
                if(cacheSz < 3) {
                    insertCache(tradesCache, qTrades);
                    return qTrades;
                } else {   //Current price is > lowest price
                    //delete lowest price record.
                    rec := getCache(tradesCache, 0);
                    deleteCache(tradesCache, 0);
                    insertCache(tradesCache, qTrades);
                    return rec;
                }
```

```
                return null;
            }
        END;

        ON QTrades {
            keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
            typeof(QTrades) rec := insertIntoCache( QTrades );
            if(rec.Id) {
                //When id does not match current id it is a
                //record to delete
                if(rec.Id <> QTrades.Id) {
                    output setOpcode(rec, delete);
                }
                output setOpcode(QTrades, upsert);
            }
        };
    END;
```

This project collects data for thirty seconds and then computes the desired output values.

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;

/* ******************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ******************************************************
 * Use Case b:
 * Perform a computation every N seconds for records
 * arrived in the last N seconds.
 *
 * Here the Nasdaq trades data is collected for 30 seconds
 * before being released for further computation.
 */
CREATE FLEX PeriodicOutputFlex
    IN QTrades
    OUT OUTPUT WINDOW QTradesPeriodicOutput SCHEMA TradesSchema
PRIMARY KEY(Symbol,Price)
    BEGIN
        DECLARE
            dictionary(typeof(QTrades), integer) cache;
                END;
        ON QTrades {
```

```
                //Whenever a record arrives just insert into
dictionary.
              //The key of the dictionary is the key to the record.
            cache[QTrades] := 0;
        };
        EVERY 30 SECONDS {
                //Cycle through event cache and output all the rows
                //and delete the rows.
                for (rec in cache) {
                        output setOpcode(rec, upsert);
                }
                clear(cache);
        };
    END;

/**
 * Perform a computation from the periodic output.
 */
CREATE OUTPUT WINDOW QTradesSymbolStats
PRIMARY KEY DEDUCED
AS SELECT
    q.Symbol,
    MIN(q.Price)        Minprice,
    MAX(q.Price)        MaxPrice,
    sum(q.Shares * q.Price)/sum(q.Shares) Vwap,
    count(*) TotalTrades,
    sum(q.Shares) TotalVolume
FROM
    QTradesPeriodicOutput q
GROUP BY
    q.Symbol
;
```

CHAPTER 2 Statements

SPLASH has statement forms for expressions, blocks, conditionals, output, “break” and “continue”, “while” and “for” loops, as well as blocks of statements.

Expression Statements

You can turn any expression into a statement by terminating the expression with a semicolon.

For example:

```
setOpcode(input, 3);
```

Since assignments are expressions, assignments can be turned into statements in the same way. For instance, the following statement assigns a string to a variable “address”:

```
address := '550 Broad Street';
```

Block Statements

Statements can be a sequence of statements, wrapped in braces, with optional variable declarations.

For example:

```
{
  float d := 9.99;
  record.b := d;
}
```

You can intersperse variable declarations with statements:

```
{
  float pi := 3.14;
  print (string(pi));
  float e := 2.71;
  print (string(e));
}
```

Conditional Statements

Use conditional statements to specify an action based on whether a specific condition is true or false. Conditional statements in SPLASH use the same syntax as conditional statements in C.

For example:

```
if (record.a = 9)
  record.b := 9.99;
```

Conditionals may have optional “else” statements:

```
if (record.a = 9)
  record.b := 9.99;
else {
  float d := 10.9;
  record.b := d;
}
```

Output Statements

The `output` statement schedules an event to be sent to downstream streams, and also to be entered into the associated store.

For example:

```
output [k = 10; | d = 20;];
```

If a Flex operator is sending output to a stateless store, all attempts to `output` a non-insert are rejected.

Note: You can use multiple output statements to process an event; the outputs are collected as a transaction block. Similarly, if a Flex operator receives a transaction block, the entire transaction block is processed and all output is collected into another transaction block. This means that downstream streams, and the record data stored within the stream, are not changed until the entire event (single event or transaction block) is processed.

While Statements

While statements are a form of conditional processing. Use them to specify an action to take while a certain condition is met. While statements use the same syntax as while statements in C and are processed as loops.

For example:

```
while (not(isnull(record))) {
  record.b := record.a + record.b;
  record := getNext(record_iterator);
}
```

For Loops

Loops are often coded with “for” loops, which provide a convenient means of looping over some or all of the records in an input stream, or all of the data in a vector or dictionary.

To loop over every record in an input stream called “input_stream”:

```
for (record in input_stream) {
  ...
}
```

The variable `record` is a new variable; you can use any name here. The scope is the statement or block of statements in the loop; it has no meaning outside the loop. You can also set equality criteria in searching for records with certain values of fields. For example:

```
for (record in input_stream where c=10, d=11) {
  ...
}
```

This statement has the same looping behavior, except limited to the records whose `c` field is 10 and `d` field is 11. If you search on the key fields, the loop runs at most one time, but it will run extremely fast because it will use the underlying index of the stream.

To loop over the values in a vector “vec1”where `val` is any new variable:

```
for (val in vec1) {
 ...
}
```

The loop stops when the end of the vector is reached, or the value of the vector is null.

To loop over the values in a dictionary “dict1” where `key` is any new variable:

```
for (key in dict1) {
 ...
}
```

It is common, inside the loop, to use the expression `dict1[key]` to get the value held in the dictionary for that particular key.

Control Statements

Use control statements to terminate or restart both `while` loops and `for` loops.

A `break` statement terminates the innermost loop; a `continue` statement starts the innermost loop over.

The `return` statement stops the processing and returns a value. This is most useful in SPLASH functions.

The `exit` statement stops the processing.

Switch Statements

The `switch` statement is a specialized form of conditional.

For instance, you can write:

```
switch(intvar*2) {
  case 0: print('case0'); break;
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

This statement prints “case0” if the value of `intvar*2` is 0, “case2” if the value of `intvar*2` is 2, and “default” otherwise. The `default` is optional. The expression inside the parentheses `switch(...)` must be of base type, and the expressions following the `case` keyword must have the same base type.

As in C and Java, the `break` is needed to skip to the end. For instance, if you leave out the `break` after the first `case`, then this statement will print both “case0” and “case2” when `intvar*2` is 0:

```
switch(intvar*2) {
  case 0: print('case0');
  case 1+1: print('case2'); break;
```

```
  default: print('default'); break;
}
```

CHAPTER 3 Data Structures

SPLASH may store and organize data in various sets of data structures designed to support specific data manipulation functions.

Record Events

SPLASH directly creates record events, which are records with an associated operation like “insert”. This documentation uses the term “record” interchangeably with “record event”.

Record Event Details

The syntax of record types specifies the names and types of fields, and the key structure.

```
[ integer key1; string key2; | string dataField; ]
```

For instance, the above type describes records with key fields key1 and key2, of types integer and string respectively, and the non-key field dataField of type string. The key fields appear before the “|” symbol.

The syntax of record values mirrors that of record types. Here is a record with the previous type:

```
[ key1 = 9; key2 = 'USD'; | string data = 'US Currency'; ]
```

The syntax of record values is fairly flexible. You can write the same record as

```
[ key1 = 9; key2 = 'USD' | string data = 'US Currency' ]
[ key1 = 9; key2 = 'USD' | string data = 'US Currency'; ]
```

eliminating any semi-colon but those between `field = value`.

The operation of a new record value is insert. To change it, use the `setOpcode` function, as in

```
setOpcode([ key1 = 9 | string data = 'US Currency' ], update)
```

Records with more fields can be used in a context expecting fewer fields; the extra fields get coerced away. Conversely, records with fewer fields can be used in a context expecting more fields; the missing fields are assumed to be null. For instance, if `var` is a variable of type

```
[ integer key1; | string dataField; float otherData]
```

you can set

```
var := [key1 = 1; dataField = 'newdata'];
```

The record value is implicitly cast to the right type, making key1 the key field and setting the otherData field to null.

Operations on records:

- **Get a field –** Syntax: `record.field`

  Type: The value returned has the type of the field.

  Example: `rec.data1`
- **Assign a field –** Assign a field in a record.

  Syntax: `record.field := value`

  Type: The value must be a value matching the type of the field of the record. The expression returns a record.

  Example: `rec.data1 := 10`
- **`getOpcode` –** Gets the operation associated with a record. The operations are of type integer, and have the following meaning:
  - 1 means “insert”
  - 3 means “update”
  - 5 means “delete”
  - 7 means “upsert”(insert if not present, update otherwise)
  - 13 means “safe delete”(delete if present, ignore otherwise)

  Syntax: `getOpcode(record)`

  Type: The argument must be an event. The function returns an integer.

  Example: `getOpcode(input)`
- **`setOpcode` –** Sets the operation associated with a record; the legal opCodeNumber operations are listed in the above description for `getOpcode`.

  Syntax: `setOpcode(record,opCodenumber)`

  Type: The first argument must be a record, and the second an integer. The function returns the modified record.

  Example: `setOpcode(input,insert)`

XML Values

An XML value is a value composed of XML elements and attributes, where elements can contain other XML elements or text. XML values can be created directly or built by parsing string values. XML values cannot be stored in records, but can be converted to string representation and stored in that form.

Operations on XML Values

You can declare a variable of `xml` type and assign it to XML values:

```
xml xmlVar;
```

In addition to declaring a variable for use with XML values, you can also perform the following operations:

- **`xmlagg`** – Aggregate a number of XML values into a single value. This can be used only in aggregate streams or with event caches (see below).

  Syntax: `xmlagg(xml value)`

  Type: The argument must be an XML value. The function returns an XML value.

  Example: `xmlagg(xmlparse(stringCol))`
- **`xmlconcat`** – Concatenate a number of XML values into a single value.

  Syntax: `xmlconcat(xml value ..., xml value)`

  Type: The arguments must be XML values. The function returns an XML value.

  Example: `xmlconcat(xmlparse(stringCol), xmlparse('<t/>'))`
- **`xmlelement`** – Create a new XML data element, with attributes and XML expressions within it.

  Syntax: `xmlelement(name xmlattributes(string AS name ..., string AS name) , xml value,...,xml value)`

  Type: The names must adhere to these conventions:

  - A name is either a sequence of alphabetic characters, digits, and underscore characters, or a sequence of any characters enclosed in double quotation marks.
  - If a name is not enclosed in double quotation marks, it must begin with an alphabetic character or an underscore.
  - A name cannot contain spaces unless it is enclosed in double quotation marks.
  - A name cannot be a Reserved Word unless it is enclosed in double quotation marks. Reserved words are case insensitive, so for example, a name cannot be “AND” or “and” or “AnD”.
  - Columns cannot be named “rowid” or “rowtime”.

The function returns an XML value.

Example: `xmlelement(top, xmlattributes('data' as attr1), xmlparse('<t/>'))`

- **`xmlparse`** – Convert a string to an XML value.

  Syntax: `xmlparse(string value)`

  Type: The argument must be a string value. The function returns an XML value.

  Example: `xmlparse('<tag/>')`
- **`xmlserialize`** – Convert an XML value to a string.

  Syntax: `xmlserialize(xml value)`

  Type: The argument must be an XML value. The function returns a string.

  Example: `xmlserialize(xmlparse('<t/>'))`

Vectors

A vector is a sequence of values, all of which must have the same type, with an ability to access elements of the sequence by an integer index. A vector has a size, from a minimum of 0 to a maximum of 2 billion entries.

Semantics and Operations

Vectors use semantics inherited from C: when accessing elements by index, index 0 is the first position in the vector, index 1 is the second, and so forth.

You can declare vectors in Global or Local blocks via the syntax:

```
vector(valueType) variable;
```

For instance, you can declare a vector holding 32-bit integers:

```
vector(integer) pos;
```

Operations on vectors:

- **Create –** Create a new empty vector.

  Syntax: `new vector(type)`

  Type: A vector of the declared type is returned.

  Example: `pos := new vector(integer);`
- **Get value by index –** Get a value from the vector. If the index is less than 0 or greater than or equal to the size of the vector, return null.

  Syntax: `vector[index]`

  Type: The index must have type integer. The value returned has the type of the values held in the vector.

Example: `pos[10]`

- **Assign a value –** Assign a cell in the vector.

  Syntax: `vector[index] := value`

  Type: The index must have type integer, and the value must match the value type of the vector. The value returned is the updated vector.

  Example: `pos[5] := 3`
- **`size`** – Returns the number of elements in the vector.

  Syntax: `size(vector)`

  Type: The argument must be a vector. The value returned has type integer.

  Example: `size(pos)`
- **`push_back`** – Inserts an element at the end of the vector and returns the modified vector.

  Syntax: `push_back(vector value)`

  Type: The second argument must be a value with the value type of the vector. The return value has the type of the vector.

  Example: `push_back(pos, 3)`
- **`resize`** – Resize a vector, either removing elements if the vector shrinks, or adding null elements if the vector expands.

  Syntax: `resize(vector newsize)`

  Type: The second argument must have type integer. The return value has the type of the vector.

  Example: `resize(vec1, 2)`

You can also iterate through all the elements in the vector (up to the first null element) using a "for" loop.

While dictionary and vector data structures can be defined globally, global use should be limited to reading them. Only one stream should write to a dictionary or vector data structure. And while that stream is writing, no other stream should write to or read from that data structure. The underlying objects used to manage the global dictionary or vector data structures are not thread-safe. A stream must have exclusive access to the global dictionary or vector data structure while writing. Allowing other streams to access these data structures while one stream is writing can result in server failure.

Use of these data structures should be limited to relatively static data (such as country codes) that will not need to be updated during processing, but will be read by multiple streams. Writing the data to the dictionary or vector must be completed before any streams read it.

All operations that read a global dictionary or vector should perform an isnull check, as shown in this example.

```
>typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

Dictionaries

Dictionaries are data structures that associate keys with values. They are called maps in C++ and Java, arrays in AWK, and association lists in LISP, so they are common data structures.

Declare a dictionary in a Global or Local block using the syntax:

```
dictionary(keyType, valueType) variable;
```

For instance, if you have an input stream called "input_stream", you could store an integer for distinct records as

```
dictionary(typeof(input_stream), integer) counter;
```

Only one value is stored per key. It is therefore important to understand what equality on keys means. For the simple datatypes, equality means the usual equality, for example, equality on integer or on string values. For record types, equality means that the keys match (the data fields and operation are ignored).

While dictionary and vector data structures can be defined globally, global use should be limited to reading them. Only one stream should write to a dictionary or vector data structure. And while that stream is writing, no other stream should write to or read from that data structure. The underlying objects used to manage the global dictionary or vector data structures are not thread-safe. A stream must have exclusive access to the global dictionary or vector data structure while writing. Allowing other streams to access these data structures while one stream is writing can result in server failure.

Use of these data structures should be limited to relatively static data (such as country codes) that will not need to be updated during processing, but will be read by multiple streams. Writing the data to the dictionary or vector must be completed before any streams read it.

All operations that read a global dictionary or vector should perform an isnull check, as shown in this example.

```
>typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

Operations on Dictionaries

Dictionaries are data structures that associate keys with values. You can perform specific operations on dictionaries.

Operations

You can perform the following operations on dictionaries:

- **Create –** Create a new empty dictionary.

  Syntax: `new dictionary(type, type)`

  Type: A vector of the declared type is returned.

  Example: `d := new dictionary(integer, string);`
- **Get value by key –** Get a value from the dictionary by key. If there is no such key in the dictionary, return null.

  Syntax: `dictionary[key]`

  Type: The key must have the type of the keys of the dictionary. The function returns a value of the type of the values held in the dictionary.

  Example: `counter[input]`
- **Assign a value by key –** Associate a value with a key in the dictionary.

  Syntax: `dictionary[key] := value`

  Type: The key and value must match the key type and value type of the dictionary. The function returns the updated dictionary.

  Example: `counter[input] := 3`
- **Remove a key/value pair –** Remove a key, and its associated value, from the dictionary.

  Syntax: `remove(dictionary, key)`

  Type: The key must match the key type of the dictionary. The function returns an integer: 0 if the key was not present, and 1 otherwise.

  Example: `remove(counter, input)`
- **Clear a dictionary –** Remove all key/value pairs from the dictionary.

  Syntax: `clear(dictionary)`

  The function returns the cleared dictionary.

  Example: `clear(counter)`
- **Test for emptiness –** Test a dictionary for emptiness.

  Syntax: `empty(dictionary)`

  The function returns an integer: 1 if the dictionary is empty, 0 if not empty.

  Example: `empty(counter)`

You can also iterate through all the key/value pairs in the dictionary using a “for” loop.

Streams

There are ways to access the records in input streams, using means similar to dictionaries, although one cannot change the records in an input stream.

Operations on Streams

- **Get value by key –** Get a value from the stream by key. If there is no such key in the stream, return null.

  Syntax: `streamValue[ recordValue ]`

  Type: The key must have the record type of the stream. The operation returns a value of the record type of the stream.

  Example: `input_stream[ [k = 3; | ] ]`

  Note: Non-key fields of the argument do not matter. The operation returns a record with the current values of the non-key fields, if a record with the key fields exists.

  If a key field is missing from the argument, or the key field is null, then this operation always returns null. It doesn't make sense to compare key fields in the stream to null, since null is never equivalent to any value (including null).
- **Get value by match –** Get a record from the stream that matches the given record. Unlike getting a value by key, there might be more than one matching record. If there is more than one matching record, one of the matching records is returned. If there is no such match in the stream, null is returned.

  Syntax: `streamValue{ recordValue }`

  Type: The record must be consistent with the record type of the stream. The operation returns a value of the record type of the stream.

  Example: `input_stream{ [ | d = 5 ] }`

  You can use key and non-key fields in the record.

You can also iterate through all the records in a stream using a “for” loop.

Stream Iterators

Stream iterators are a means of explicitly iterating over all of the records stored in a stream. It is usually more convenient, and safer, to use the `for` loop mechanism, but stream iterators provide extra flexibility.

Functions for Stream Iterators

Each block of code has implicit variables for streams and stream iterators. If an input stream is named `Stream1`, there are variables `Stream1_stream` and `Stream1_iterator`.

Those variables can be used in conjunction with the following functions.

- **`deleteIterator`** – Releases the resources associated with an iterator.

  Syntax: `deleteIterator(iterator)`

  Type: The argument must be an iterator expression. The function returns a null value.

  Example: `deleteIterator(input_iterator)`

  Note: Stream iterators are not implicitly deleted. If you don't delete them explicitly, all further updates to the stream may be blocked.

- **`getIterator`** – Get an iterator for a stream.

  Syntax: `getIterator(stream)`

  Type: The argument must be a stream expression. The function returns an iterator.

  Example: `getIterator(input_stream)`

- **`getNext`** – Returns the next record in the iterator, or null if there are no more records.

  Syntax: `getNext(iterator)`

  Type: The first argument must be an iterator expression. The function returns a record, or "null" if there is no more data in the iterator.

  Example: `getNext(input_iterator)`

- **`resetIterator`** – Resets the iterator to the beginning.

  Syntax: `resetIterator(iterator)`

  Type: The argument must be an iterator expression. The function returns an iterator.

  Example: `resetIterator(input_iterator)`

- **`setRange`** – Sets a range of columns to search for. Subsequent getNext calls return only those records whose columns match the given values.

  Syntax: `setRange(iterator fieldName... expr...)`

Type: The first argument must be an iterator expression; the next arguments must be the names of fields within the record; the final arguments must be expressions. The function returns an iterator.

Example: `setRange(input_iterator,Currency,Rate,'EUR',9.888)`

- **`setSearch`** – Sets values of columns to search for. Subsequent getNext calls return only those records whose columns match the given values.

  Syntax: `setSearch( iterator number... expr...)`

  Type: The first argument must be an iterator expression; the next arguments must be column numbers (starting from 0) in the record; the final arguments must be expressions. The function returns an iterator.

  Example: `setSearch(input_iterator,0,2,'EUR',9.888)`

Note: The `setSearch` function has been deprecated because it requires a specific layout of fields. It has been retained for backwards compatibility with existing projects. When developing new projects, use the `setRange` function instead.

Event Caches

Event caches hold a number of previous events for the input stream or streams to a derived stream. They are organized into buckets, based on values of the fields in the records and are often used when vectors or dictionaries are not quite the right data structure.

You can define an event cache in a Local block. A simple event cache declaration:

```
eventCache(input_stream) e0;
```

This event cache holds all the events for an input stream “input_stream”. The default key structure of the input stream defines the bucket policy. That is, the buckets in this stream correspond to the keys of the input stream.

Suppose the input stream in this case has two fields, a key field `k` and a data field `d`. Suppose the events have been:

```
<input_stream ESP_OPS="i" k="1" d="10"/>
<input_stream ESP_OPS="u" k="1" d="11"/>
<input_stream ESP_OPS="i" k="2" d="21"/>
```

After these events have flowed in, there will be two buckets. The first bucket will contain the first two events, because these have the same key; the second bucket will contain the last event.

Event caches allow for aggregation over events. That is, the ordinary aggregation operations that can be used in aggregate streams can be used in the same way over event caches. The “group” that is selected for aggregation is the one associated with the current event.

```
<input_stream ESP_OPS="u" k="1" d="12"/>
```

For instance, if the above event appears in this stream, then the expression `sum(e0.d)` returns 10+11+12=33. You can use any of the accepted aggregation functions, including `avg`, `count`, `max`, and `min`.

Manual Insertion

By default, every event that comes into a stream with an event cache gets put into the event cache.

You can explicitly indicate this default behavior with the `auto` option:

```
eventCache(instream, auto) e0;
```

You can also put events into an event cache if they are marked `manual`:

```
eventCache(instream, manual) e0;
```

Use the function `insertCache` to do this.

Changing Buckets

An event cache organizes events into buckets. By default, the buckets are determined from the keys of the input stream. You can change that default behavior to alternative keys, specifying other fields in square brackets after the name of the stream.

Specifying the following keeps buckets organized by distinct values of the `d0` and `d1` fields:

```
eventCache(instream[d0,d1]) e0;
```

To keep one large bucket of all events, write the following:

```
eventCache(instream[]) e0;
```

Managing Bucket Size

You can manage the size of buckets in an event cache. That can often be important in controlling the use of memory.

You can limit the size of a bucket to the most recent events, by number of seconds, or by time:

```
eventCache(instream, 3 events) e0;
eventCache(instream, 3 seconds) e1;
```

You can also specify whether to completely clear the bucket when the size or time expires by specifying the `jump` option:

```
eventCache(instream, 3 seconds, jump);
```

The default is `nojump`.

All of these options can be used together. For example, this example clears out a bucket when it reaches 10 events (when the 11th event comes in) or when 3 seconds elapse.

```
eventCache(instream, 10 events, 3 seconds, jump);
```

Keeping Records

You can keep records in an event cache, instead of distinct events for insert, update, and delete, by specifying the `coalesce` option.

For example:

```
eventCache(instream, coalesce) e0;
```

This option is most often used in conjunction with the ordering option.

Ordering

Normally, the events in a bucket are kept by order of arrival. You can specify a different ordering by the fields of the events.

For instance, to keep the events in the bucket ordered by field `d` in descending order:

```
eventCache(instream, d desc) e0;
```

You can order by more than one field. The following example orders the buckets by field `d0` in descending order, then by field `d1` in ascending order in case the `d0` fields are equal.

```
eventCache(instream, d0 desc, d1 asc) e0;
```

Operations on Event Caches

Event caches hold a number of previous events for the input stream or streams to a derived streams.

Supported Event Cache Operations

- **`expireCache`** – Remove events from the current bucket that are older than a certain number of seconds.

Syntax: `expireCache(events, seconds)`

Type: The first argument must name an event cache variable. The second argument must be an integer. The function returns the event cache.

Example: `expireCache(events, 50)`

- **`insertCache`** – Insert a record value into an event cache.

  Syntax: `insertCache(events, record)`

  Type: The first argument must name an event cache variable. The argument must be a record type. The function returns the record inserted.

  Example: `insertCache(events, inputStream)`

- **`keyCache`** – Select the current bucket in an event cache. Normally, the current input record selects the active bucket. You might want to change the current active bucket in some cases. For example, during the evaluation of the debugging expressions, there is no current input record and thus no bucket is set by default. The only way to set the bucket then is to do it manually using this function.

  Syntax: `keyCache(events, event)`

  Type: The first argument must name an event cache variable. The second argument must be a record type. The function returns the same record.

  Example: `keyCache(ec1, rec)`

- **`getCache`** – Returns the row specified by a given index from the current bucket in the event cache. This index is 0 based. The function takes an integer as its argument, and the function returns a row. Specifying an invalid index parameter will result in the generation of a bad record.

  Syntax: `getCache(cache, index)`

  Type: The first argument must name an event cache variable. The second argument must be an integer specifying the row to retrieve. The function returns the specified row of the cache.

  Example: `getCache(tradesCache, 3)`

- **`deleteCache`** – Returns the row specified by a given index from the current bucket in the event cache. This index is 0 based. The function takes an integer as its argument, and the function returns a row. Specifying an invalid index parameter will result in the generation of a bad record.

  Syntax: `deleteCache(cache, index)`

  Type: The first argument must name an event cache variable. The second argument must be an integer specifying the row to delete. The function deletes the specified row; it does not return any output.

  Example: `deleteCache(tradesCache, 0)`

- **`cacheSize` –** Returns the size of the current bucket in the event cache.

  Syntax: `cacheSize()`

  Type: This function takes no arguments. It returns an integer.

  Example: `cacheSize()`

Index