SYBASE®
An SAP® Company

SPLASH Tutorial

Sybase Event Stream Processor 5.0

DOCUMENT ID: DC01719-01-0500-02
LAST REVISED: December 2011

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

Contents

CHAPTER 1 Introduction

CCL, the language used by the Sybase® Event Stream Processor, has two components.

The first component is an extension to SQL, which is used to declare the streams and other high level components, describe the data flow and define logic that can be expressed as relational operations (such as joins, aggregations, filters, etc.). The second component is SPLASH, which is a procedural language that is used to specify logic that cannot be easily represented using relational operators. SPLASH, just like any other procedural language, has support for variables, looping constructs, data structures and defining functions.

CHAPTER 2 Basics

Understand the basics of the language before dealing with functions and advanced data structures.

Although SPLASH is similar to C/C++ and Java, it has its own subtleties you should be aware of.

Constants and Simple Expressions

Both string constants and numeric expressions are primitives in SPLASH.

Take the SPLASH version of the classic "hello world" program as an example:

```
print('hello world\n');
```

This statement prints the line `hello world` on the console of the Sybase ESP Studio or on the terminal where the Sybase ESP Server was started. This sample code contains a string constant, surrounded by single quotes. This differs from C/C++ and Java, which use double quotes for string constants.

Numeric expressions are another primitive part of the language. Numeric constants are in integer form (1826), floating point decimal form (72.1726), or fixed-point decimal form. You can form expressions from the standard arithmetic operators (+, -, *, /, ^) and parentheses, with precedence defined as usual. The following expression will compute 57:

```
1 + 7 * 8
```

However, this expression will compute 64:

```
(1 + 7) * 8
```

SPLASH includes a host of arithmetic functions too. This function returns the sine of the value:

```
sine(1.7855)
```

Like C/C++ and Java, boolean expressions—that return true or false—are represented by numeric expressions. The integer 0 represents false, and any non-0 integer represents true. Comparison operators like = (equal), != (not equal), < (less than), and > (greater than) return 0 if false, and 1 if true. You can use the operators `and`, `or`, and `not` to combine boolean expressions. For instance, `not(0 >1)` returns 1.

Null Values

Each datatype contains a distinguished empty value, written `null` for its correspondence with null values in relational databases. The null value encodes a missing value. It cannot be compared to any value, including itself. Thus, the expressions `(null = null)` and `(null != null)` are both 0 (false).

Most built-in functions return null when given null. For instance, `sqrt(null)` returns null.

Since you cannot compare values to null, there are special SPLASH functions for handling null values. The function `isnull` returns 1 (true) if its argument is null, and 0 (false) otherwise. Because it is common to want to choose a non-null value among a sequence of values, there is another function, `firstnonnull`, whose return value is the first non-null value in the sequence of values. The following example returns 3:

```
firstnonnull(null,3,4,5)
```

Variables and Assignment

As with other programming languages, using variables within SPLASH involves declaring the variables and assigning them values.

For example, the following code declares a variable named `eventCount` that holds values of type integer (32-bit integers):

```
integer eventCount;
```

When you have declared the variable, assign it a value using the `:=` operator:

```
eventCount := 4;
```

Use the value of the variable by writing its name:

```
eventCount + 1
```

To be concise, you can mix declarations and assignments to initial values:

```
float pi := 3.14159265358979;
money dollarsPerEuro := 1.58d;
```

You can also declare multiple variables of the same type in a single declaration, even mixing those with initial values and those without. This declaration describes three variables, all of datatype float, with `pi` and `e` set to initial values:

```
float pi := 3.14159265358979, lambda, e := 2.714;
```

Variables that begin with non-alphabetic characters or keywords in SPLASH can be turned into variable names with double quotes:

```
long "500miles" := 500 * 1760;
string "for" := 'for ever';
```

This feature comes directly from SQL.

Explicit variable declarations make SPLASH into a statically typed language like C and Java. This is different than scripting languages like Perl and AWK, which do not force the declaration of variables. At the cost of more characters, it makes the code easier to maintain and allows optimizations that make the code run faster.

Datatypes

SPLASH uses the standard CCL datatypes integer, string, float, long, money, money(n), date, timestamp, bigdatetime, interval, binary, and boolean.

Datatype	Description
`integer`	A signed 32-bit integer. The range of allowed values is -2147483648 to +2147483647 (-2^{31} to 2^{31-1}). Constant values that fall outside of this range are automatically processed as long datatypes. To initialize a variable, parameter, or column with a value of -2147483648, specify (-2147483647) -1 to avoid compiler errors.
`long`	A signed 64-bit integer. The range of allowed values is -9223372036854775808 to +9223372036854775807 (-2^{63} to 2^{63-1}). To initialize a variable, parameter, or column with a value of -9223372036854775807, specify (-9223372036854775807) -1 to avoid compiler errors.
`float`	A 64-bit numeric floating point with double precision. The range of allowed values is approximately -10^{308} through $+10^{308}$.
`string`	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes.
`money`	A signed 64-bit integer with global precision. Currency symbols and commas are not supported in the input data stream.

Datatype	Description
`money(n)`	A signed 64-bit integer that supports varying precision, from 1 to 15 digits after the decimal point. Currency symbols and commas are not supported in the input data stream, however, decimal points are. The supported range of values change, depending on the specified precision. `money(1)`: -922337203685477580.8 to 922337203685477580.7 `money(2)`: -92233720368547758.08 to 92233720368547758.07 `money(3)`: -9223372036854775.808 to 9223372036854775.807 `money(4)`: -922337203685477.5808 to 922337203685477.5807 `money(5)`: -92233720368547.75808 to 92233720368547.75807 `money(6)`: -92233720368547.75808 to 92233720368547.75807 `money(7)`: -922337203685.4775808 to 922337203685.4775807 `money(8)`: -92233720368.54775808 to 92233720368.54775807 `money(9)`: -9223372036.854775808 to 9223372036.854775807 `money(10)`: -922337203.6854775808 to 922337203.6854775807 `money(11)`: -92233720.36854775808 to 92233720.36854775807 `money(12)`: -9223372.036854775808 to 9223,372.036854775807 `money(13)`: -922337.2036854775808 to 922337.2036854775807 `money(14)`: -92233.72036854775808 to 92233.72036854775807 `money(15)`: -9223.372036854775808 to 9223.372036854775807 To initialize a variable, parameter, or column with a value of -92,233.72036854775807, specify (-9...7) -1 to avoid compiler errors. Specify explicit precision for money constants with **Dn** syntax, where **n** represents the precision. For example, 100.1234567D7, 100.12345D5. Implicit conversion between `money(n)` types is not supported because there is a risk of losing range or precision. Perform the **cast** function to work with money types that have different precision.

Datatype	Description
`bigdatetime`	Timestamp with microsecond precision. The default format is YYYY-MM-DDTHH:MM:SS:SSSSSS. All numeric datatypes are implicitly cast to `bigdatetime`. The rules for conversion vary for some datatypes: • All `boolean`, `integer`, and `long` values are converted in their original format to `bigdatetime` • Only the whole-number portions of `money(n)` and `float` values are converted to `bigdatetime`. Use the **cast** function to convert `money(n)` and `float` values to `bigdatetime` with precision. • All `date` values are multiplied by 1000000 and converted to microseconds to satisfy `bigdatetime` format. • All `timestamp` values are multiplied by 1000 and converted to microseconds to satisfy `bigdatetime` format.
`timestamp`	Timestamp with millisecond precision. The default format is YYYY-MM-DDTHH:MM:SSS.
`date`	Timestamp with millisecond precision. The default format is YYYY-MM-DDTHH:MM:SSS.

Datatype	Description
`interval`	A signed 64-bit integer that represents the number of microseconds between two timestamps. Specify an `interval` using multiple units in space-separated format, for example, "5 Days 3 hours 15 Minutes". External data that is sent to an interval column is assumed to be in microseconds. Unit specification is not supported for `interval` values converted to or from `string` data. When an `interval` is specified, the given interval must fit in a 64-bit integer (`long`) when it is converted to the appropriate number of microseconds. For each `interval` unit, the maximum allowed values that fit in a long when converted to microseconds are: • MICROSECONDS (MICROSECOND, MICROS): +/- 9223372036854775807 • MILLISECONDS (MILLISECOND, MILLIS): +/- 9223372036854775 • SECONDS(SECOND, SEC): +/- 9223372036854 • MINUTES(MINUTE, MIN): +/- 153722867280 • HOURS(HOUR,HR): +/- 2562047788 DAYS(DAY): +/- 106751991 The values in parentheses are alternate names for an `interval` unit. When the maximum value for a unit is specified, no other unit can be specified or it causes an overflow. Each unit can be specified only once.
`binary`	Represents a raw binary buffer. Maximum length of value is platform-dependent, but can be no more than 65535 bytes. NULL characters are permitted.
`boolean`	Value is true or false. The format for values outside of the allowed range for `boolean` is 0/1/false/true/y/n/on/off/yes/no, which is case-insensitive.

Programs in SPLASH automatically convert values of numeric type to other types if possible.

```
float e := 2.718281828459;
integer years := 10;
```

If you declare the above, then the following expression is a legal expression with datatype float:

```
1000.0d * (e ^ (0.05 * years))
```

The variable `years` of type integer is automatically converted to float, as is the constant 1000.0d of type money.

If necessary, use the cast operation to downcast values (convert a number with more precision into a number with less precision). For instance, this expression converts the float value into an integer by truncating the decimal part of the number:

```
cast(integer, 1000.0d * (e ^ (0.05 * years)))
```

You can compute with values of datatype date, timestamp and bigdatetime just as if they are numeric values. For example, if you add 10 to a date value, the result is a date value ten seconds in the future. Likewise, if you add 10 to a timestamp value, the result is a timestamp value ten milliseconds in the future. The precision is thus implied in the type.

Datatype Abbreviations

Use datatype abbreviations to provide alternate names to datatypes. Abbreviations are most useful when working with long datatype names.

To give alternative names to datatypes, use the `typedef` declaration. This example declares `euros` to be another name for the money datatype:

```
typedef money euros;
```

This declares a variable `price` of that datatype:

```
euros price := 10.70d;
```

You can also use the `typeof` operator to simplify datatype definitions. This operator returns the datatype of the expression.

```
typeof(price) newPrice := 10.70d;
```

The above is an equivalent way of writing:

```
money newPrice := 10.70d;
```

Blocks

Segment code into blocks to declare variables and set their values local to the block.

Blocks of statements are written between curly braces.

```
{
  float pi := 3.1415926;
  float circumference := pi * radius;
}
```

This declares a variable `pi` that is local to the block, and uses that variable to set a variable called `circumference`. Variable declarations (but not type abbreviations) may be interspersed with statements; they need not be at the beginning of a block.

When you nest blocks inside one another, the usual scoping rules apply. This SPLASH code prints “there” instead of “here”:

```
{
  string t := 'here';
  {
    string t := 'there';
    print(t);
```

```
  }
}
```

Control Structures

The control structures of SPLASH are similar to those in C and Java.

Use the `if` and `switch` statements for conditional execution. For example, this block sets a string variable to different values depending on the sign of the temperature:

```
if (temperature < 0) {
  display := 'below zero';
} else if (temperature = 0) {
  display := 'zero';
} else {
  display := 'above zero';
}
```

The `switch` statement selects from a number of alternatives:

```
switch (countryCode) {
  case 1:
    continent := 'North America';
    break;
  case 33:
  case 44:
  case 49:
    continent := 'Europe';
    break;
  default:
    continent := 'Unknown';
}
```

The expression after the `switch` can be an integer, long, money, money(n), float, date, timestamp, or bigdatetime.

The `while` statement encodes loops. For example, this computes the sum of the squares of 0 through 9:

```
integer i := 0, squares := 0;
while (i < 10) {
  squares := squares + (i * i);
  i++;
}
```

This example uses the operator `++` to add 1 to the variable `i`. The `break` statement exits the loop, and `continue` starts the loop again at the top.

Finally, you can stop the execution of a block of SPLASH code with the `exit` statement. This statement doesn't stop the Event Stream Processor, just the block.

CHAPTER 3 Record Events

In the Event Stream Processor, streams process “record events.” An event is a record, which is a composite value that associates field names with values, and an operation (for example, insert or update). Some of the fields may be designated as key fields.

Record Types

A record type defines the structure of the record, including the name and datatype of each field in the record.

Here's an example of a record type:

```
[ string Symbol; | integer Shares; float Price; ]
```

The type describes a record with three fields: a string field called Symbol, which is the sole key field because it is to the left of the `|` symbol; an integer field representing the number of shares; and a float field representing the price.

Each field must have one of the basic datatypes (integer, long, float, money, money(n), string, date, bigdatetime, timestamp, interval, binary, or boolean). Records cannot be nested.

Because record types are long, it is often helpful to use `typedef` to give them a shorter name:

```
typedef [ string Symbol; | integer Shares; float Price; ] rec_t;
```

The `typedef` in this example creates a record with three fields (the same as in the first example) and gives it the name `rec_t`.

Record Values

When you have a named record, you can assign it values - either static or variable.

Two record values of type `rec_t` are

```
[ Symbol='T'; | Shares=10; Price=20.15; ]
[ Symbol='GM'; | Shares=5; Price=16.81; ]
```

You can assign a record variable. The following declares a record variable and assigns a record value to it:

```
rec_t rec := [ Symbol='T'; | Shares=10; Price=22.88; ];
```

To get the value of a field in a record, you use the “.” operator. For instance, the expression `rec.Symbol` returns the string “T”.

Record values can be null. An attempt to access a field in a null record returns null.

You can change the value of a field in a record without having to recreate a new one. This example changes the value of the Shares field to 80:

```
rec.Shares := 80;
```

Key Fields

Streams store at most one record for each unique key. That is, the values in the key field or fields must be unique within the stream.

The following records each have unique keys:

```
[ Market = 'NYSE'; Symbol='T'; | Shares=10; Price=22.88; ]
[ Market = 'NYSE'; Symbol='GM'; | Shares=5; Price=16.81; ]
```

This third record matches the first record:

```
[ Market = 'NYSE'; Symbol='T'; | Shares=10; Price=20.15; ]
```

You cannot store both inside a stream, although you can overwrite the first record with this one with an update.

Record Casting

Records are implicitly coerced depending on their context. Extra fields are dropped and missing fields are made null.

For instance, this assignment drops the Extra field before the assignment is made:

```
rec_t rec := [ Symbol='T'; | Shares=10; Price=22.88; Extra=1; ];
```

Conversely, this assigns the variable `rec` to a record whose Price field is null:

```
rec_t rec := [ Symbol='T'; | Shares=10; ];
```

SPLASH is also forgiving about the key fields. For instance, if you forget to make the Symbol field a key field in the following, it will make the Symbol field into a key field:

```
rec_t rec := [ | Symbol='T'; Shares=10; Price=22.88; Extra=1; ];
```

Key fields should not be null. It is legal to assign teh following but you cannot send this to downstream streams:

```
rec_t rec := [ | Shares=10; Price=22.88; Extra=1; ];
```

This will be described in more detail in later topics.

Hidden Fields

Records have three implicit fields, `ROWID`, `ROWTIME` and `BIGROWTIME`, of datatype long, date and bigdatetime respectively. Streams fill in these values automatically, and you can access them with the usual "." operation.

Operations

Implicit within each event is an operation that is either insert, update, delete, upsert, or safedelete.

Each operation has an equivalent numeric code, and there are special constants `insert`, `update`, `delete`, `upsert`, and `safedelete` for these numeric values.

- "insert" means insert a record. It's a run-time error if there is already a record present with those keys.
- "update" means update a record. It's a run-time error if there is no record present with those keys.
- "delete" means delete a record. It's a run-time error if there is no record present with those keys.
- "upsert" means insert a record if no record is present with those keys and update otherwise. This avoids the potential run-time error with "insert" or "update."
- "safedelete" means delete a record if one is already present with those keys and ignore otherwise. This avoids the potential run-time error with "delete."

The operation is set to insert when a record event is created.

Use the function `getOpcode` to get the operation out of an event, and `setOpcode` to set the operation. The function `setOpcode` alters the record event without making a copy. For instance, this prints the numeric codes for insert (which is 1) and safedelete (which is 13):

```
[ integer k; | string data;] v;
v := [k=9;|];
print('opcode=', string(getOpcode(v)), '\n');
setOpcode(v,safedelete);
print('opcode=', string(getOpcode(v)), '\n');
```

The operations within record events are used in streams and event caches. This is described in more detail in subsequent topics.

CHAPTER 4 SPLASH Functions

CCL contains a large number of built-in functions, all of which can be used in SPLASH expressions. You can also write your own functions in SPLASH. They can be declared in global blocks for use by any stream, or local blocks for use in one stream. Functions can internally call other functions, or call itself recursively.

Note: You can also write your own functions in C/C++ or Java. Consult the reference information for details on building libraries and calling them from within the Event Stream Processor.

The syntax for declaring SPLASH functions resembles C. In general, a function looks like

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

where each “type” is a SPLASH type, and each `arg` is the name of an argument. The body of the function is a block of statements, which can start with variable declarations. The value returned by the function is the value returned by the `return` statement within.

Here is an example of a recursive function:

```
integer factorial(integer x) {
   if (x <= 0) {
     return 1;
   } else {
     return factorial(x-1) * x;
   }
 }
```

Here is an example of two mutually recursive functions (a particularly inefficient way to calculate the evenness or oddness of a number):

```
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);
   }
}
```

Unlike C, you do not need a prototype of the “even” function in order to declare the “odd” function.

The next two functions illustrate multiple arguments and record input.

```
integer sumFun(integer x, integer y) {
  return x+y;
}
string getField([ integer k; | string data;] rec) {
  return rec.data;
}
```

The real use of SPLASH functions is to define a computation once. Suppose 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 a function and use it in many places within the project:

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

CHAPTER 5 Advanced Data Structures

SPLASH allows you to store data inside data structures for later use. There are three main types: vectors, dictionaries, and event caches.

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
}
```

Vectors

A vector is a sequence of values, all of the same type. It's like an array in C, except that the size of a vector can be changed at run time.

The following block creates a new vector storing the “roots of unity” without the imaginary component.

```
vector(float) roots;
integer i := 0;
float pi := 3.1415926, e := 2.7182818, sum1 := 0;
resize(roots, 8);  // new size is 8, with each element set to null
while (i < 8) {
  roots[i] := e ^ ((pi * i) / 4);
  i++;
}
```

It creates an empty vector, resizes it with `resize`, and assigns values to elements in the vector. The first element of the vector has index 0, the second index 1, and so forth. You can also add new elements to the end of a vector with the `push_back` operator, for example, `push_back(roots, e^pi)`.

Here's a way to calculate the sum of the values of the `roots` vector:

```
i := 0;
while (i < size(roots)) {
  sum1 := sum1 + roots[i];
  i++;
}
```

The `size` operation returns the size of the vector. You can also loop through the elements of a vector using a `for` loop:

```
for (root in roots) {
  sum1 := sum1 + root;
}
```

The variable `root` is a new variable whose scope is restricted to the loop body. The first time through the loop, it is `roots[0]`, the second time `roots[1]`, and so forth. The loop stops when `roots[n]` is null or there are no more elements in `roots`.

Two other operations on vectors are useful. You can create a new vector with the `new` operation:

```
roots := new vector(float);
```

The old vector is automatically thrown away.

Dictionaries

A dictionary associates keys with values. Keys and values can have any type, which makes dictionaries more flexible than vectors at the cost of slightly slower access.

Here's an example that creates and initializes a dictionary of currency conversion rates:

```
dictionary(string, money) convertFromUSD;
convertFromUSD['EUR'] := 1.272d;
convertFromUSD['GBP'] := 1.478d;
convertFromUSD['CAD'] := 0.822d;
```

Only one value per distinct key can be held, so this statement overwrites the previous value associated with the key “EUR”:

```
convertFromUSD['EUR'] := 1.275d;
```

The expression `convertFromUSD['CAD']` extracts the value from the dictionary. If there is no matching key, as in `convertFromUSD['JPY']`, the expression returns null.

You can use the function `remove` to remove a key and its value from a dictionary. For instance, `remove(convertFromUSD,'EUR')` removes the key and corresponding value for Euros. The function `clear` removes all keys from the dictionary. You can test whether the dictionary has no more keys with the `empty` operation.

To loop through all elements in the dictionary, you can use a `for` loop:

```
for (currency in convertFromUSD) {
  if (convertFromUSD[currency] > 1) {
    print('currency ', currency, ' is worth more than one USD.\n');
  }
}
```

The variable `currency`, whose scope is restricted to the loop body, has the type of keys of the dictionary (string in this case).

Finally, you can create new dictionaries with the `new` operation. For instance, this creates an empty dictionary and assigns it to `convertFromUSD`:

```
convertFromUSD := new dictionary(string, money);
```

Mixing Vectors and Dictionaries, and Reference Semantics

When working with vectors and dictionaries, you can build simple or complex structures. For example, you can build vectors of vectors, or vectors of dictionaries, or any other mix.

For instance, you might want to store a sequence of previous stock prices by ticker symbol. This declaration and function creates such a set of stored prices, keyed by symbols:

```
dictionary(string, vector(money)) previousPrices;
integer addPrice(string symbol, money price)
{
  vector(money) prices := previousPrices[symbol];
  if (isnull(prices)) {
    prices := new vector(money);
    previousPrices[symbol] := prices;
  }
  push_back(prices, price);
}
```

The example relies on reference semantics of containers. For instance, this assignment returns a reference to the vector, not a copy of the vector:

```
vector(money) prices := previousPrices[symbol];
```

Because it is a reference, the value inserted by `push_back` is in the vector the next time it is read from the dictionary.

Reference semantics does permit aliasing, that is, alternative names for the same entity. For instance, this results in the program printing “aliased!”:

```
dictionary(integer, integer) d0 := new dictionary(integer, integer);
dictionary(integer, integer) d1 := d0;
d1[0] := 1;
if (d0[0] = 1) print('aliased!');
```

Event Caches

An event cache is a special SPLASH data structure for grouping and storing events from an input stream. Events are grouped into buckets. You can run aggregate operations like `count`, `sum`, and `max` over a bucket.

You declare an event cache in the Local block of a stream using the name of the input stream. For example, this declares an event cache for the input stream Trades:

```
eventCache(Trades) events;
```

You can have as many event caches as you wish per input stream, so you can declare this in the same stream:

```
eventCache(Trades) moreEvents;
```

By default, the buckets are determined by the keys of the input stream. For example, if you have an input stream Trades with two buckets, one for events with Symbol “T” and one with symbol “CSCO”:

```
[ Symbol='T'; | Shares=10; Price=22.88; ]
[ Symbol='CSCO'; | Shares=50; Price=15.66; ]
```

Each event—whether an insert, update, or delete—is put into the corresponding bucket. For instance, if a delete event carrying this bucket comes into the stream, then there are two events in the bucket for “CSCO”:

```
[ Symbol='CSCO'; | Shares=50; Price=15.66; ]
```

You can change that behavior by declaring the event cache to coalesce events:

```
eventCache(Trades, coalesce) events;
```

In this case, the bucket for “CSCO” then has no events.

You can compute over buckets through aggregate operations. For instance, if you want to compute the total number of shares in a bucket, write `sum(events.Shares)`. Which bucket gets selected? By default, it's the bucket associated with the current event that came from the input stream. You can change the bucket with the `keyCache` operation.

There are ways to change the way buckets are stored. For example, you can group events into buckets by specifying columns, say by Trades that have the same number of shares:

```
eventCache(Trades[Shares]) eventsByShares;
```

Or group the same number of shares and the same symbol:

```
eventCache(Trades[Symbol, Shares]) eventsBySymbolShares;
```

Or group in one large bucket:

```
eventCache(Trades[]) eventsAll;
```

You can also order the events in the bucket by field:

```
eventCache(Trades, Price desc) eventsOrderByPrice;
```

This orders the events by descending order of Price. You can use the `nth` operation to get the individual elements in that order.

If the input stream has many updates, buckets can become very big. You can control the size of buckets either by specifying a maximum number of events or a maximum amount of time or both. For example, you can set a maximum number of 10 events per bucket:

```
eventCache(Trades[Symbol], 10 events) eventsBySymbol10Events;
```

Or set a maximum age of 20 seconds:

```
eventCache(Trades[Symbol], 20 seconds) eventsBySymbol20Seconds;
```

Or both:

```
eventCache(Trades[Symbol], 10 events, 20 seconds) eventsSmall;
```

CHAPTER 6 Integrating SPLASH into CCL

CCL uses Flex operators to execute SPLASH code to process events. They have local declaration blocks, which are blocks of SPLASH function and variable declarations. They also have one method block per input stream and an optional timer block also written in SPLASH.

Access to the Event

When an event arrives at a Flex operator from an input stream, the method for that input stream is run.

The SPLASH code for that method has two implicitly declared variables for each input stream: one for the event and one for the old version of the event. More precisely, if the input stream is named InputStream, the variables are:

- `InputStream`, with the type of record events from the input stream, and
- `InputStream_old`, with the type of record events from the input stream.

When the method for input stream is run, the variable `InputStream` is bound to the event that arrived from that stream. If the event is an update, the variable `InputStream_old` is bound to the previous contents of the record, otherwise it is null.

Note: Delete events always come populated with the data previously held in the input stream.

A Flex operator can have more than one input stream. For instance, if there is another input stream called AnotherInput, the variables `AnotherInput` and `AnotherInput_old` are implicitly declared in the method block for InputStream. They are set to null when the method block begins, but can be assigned within the block.

Access to Input Streams

Within method and timer code in Flex operators, you can examine records in any of the input streams.

More precisely, there are implicitly declared variables:

- `InputStream_stream` and
- `InputStream_iterator`.

The variable `InputStream_stream` is quite useful for looking up values. The `InputStream_iterator` is less commonly used and is for advanced users.

For example, suppose you are processing events from an input stream called Trades, with the following records:

```
[ Symbol='T'; | Shares=10; Price=22.88; ]
```

You might have another input stream called Earnings that contains recent earnings data, storing records:

```
[ Symbol='T'; Quarter="2008Q1"; | Value=10000000.00;  ]
```

In processing events from Earnings, you can look up the most recent Trades data using:

```
Trades := Trades_stream[Earnings];
```

The record in the Trades stream that has the same key field Symbol. If there is no matching record in the Trades stream, the result is null.

When processing events from the Trades stream, you can look up earnings data using:

```
Earnings := Earnings_stream{ [ Symbol = Trades.Symbol; | ] };
```

The syntax here uses curly braces rather than square brackets because the meaning is different. The Trades event does not have enough fields to look up a value by key in the Earnings stream. In particular, it's missing the field called Quarter. The curly braces indicate "find any record in the Earnings stream whose Symbol field is the same as `Trades.Symbol`". If there is no matching record, the result is null.

If you have to look up more than one record, you can use a `for` loop. For instance, you might want to loop through the Earnings stream to find negative earnings:

```
for (earningsRec in Earnings_stream) {
  if ( (Trades.Symbol = Earnings.Symbol) and (Earnings.Value < 0) ) {
    negativeEarnings := 1;
    break;
  }
}
```

As with other `for` loops in SPLASH, the variable `earningsRec` is a new variable whose scope is the body of the loop. You can write this slightly more compactly:

```
for (earningsRec in Earnings_stream where Symbol=Trades.Symbol) {
  if (Earnings.Value < 0) {
    negativeEarnings := 1;
    break;
  }
}
```

This loops only over the records in the Earnings stream that have a Symbol field equal to `Trades.Symbol`. If you happen to list the key fields in the `where` section, the loop runs very efficiently. Otherwise, the `where` form is only nominally faster than the first form.

Using a Flex operator, you can access records in the stream itself. For instance, if the Flex operator is called Flex1, you can write a loop just as you can with any of the input streams:

```
for (rec in Flex1) {
  ...
}
```

Output Statement

Typically, a Flex operator method creates one or more events in response to an event. In order to use these events to affect the store of records, and to send downstream to other streams, use the `output` statement.

Here's code that breaks up an order into ten new orders for sending downstream:

```
integer i:= 0;
while (i < 10) {
  output setOpcode([Id = i; |
                    Shares = InStream.Shares/10;
                    Price = InStream.Price; ], upsert);
}
```

Each of these is an upsert, which is a particularly safe operation; it gets turned into an insert if no record with the key exists, and an update otherwise.

Notes on Transactions

A Flex operator method processes one event at a time. The Event Stream Processor can, however, be fed data in transaction blocks (groups of insert, update, and delete events).

In such cases, the method is run on each event in the transaction block. The Event Stream Processor maintains an invariant: a stream takes in a transaction block, and produces a transaction block. It's always one block in, one block out. The Flex operator pulls apart the transaction block, and runs the method on each event within the block. All of the events that `output` are collected together. The Flex operator then atomically applies this block to its records, and sends the block to downstream streams.

If you happen to create a bad event in processing an event, the whole block is rejected. For example, if you try to output a record with any null key columns.

```
output [ | Shares = InStream.Shares; Price = InStream.Price; ];
```

This whole transaction block would be rejected. Likewise, if you try the following implicit insert:

```
output [Id = 4; |
        Shares = InStream.Shares;
        Price = InStream.Price; ];
```

If there is already a record in the Flex operator with Id set to 4, the block is rejected. You can get a report of bad transaction blocks by starting the Event Stream Processor with the `-B` option. Often it's better to ensure that key columns are not null, and use `setOpcode` to create upsert or safedelete events so that the transaction block is accepted.

Transaction blocks are made as small as possible before they are sent to other streams. For instance, if your code outputs two updates with the same keys, only the second update is sent downstream. If your code outputs an insert followed by a delete, both events are removed from the transaction block. Thus, you might output many events, but the transaction block might contain only some of them.

CHAPTER 7 Sample SPLASH Code

Reviewing samples of SPLASH code is the best way to familiarize yourself with its constructs.

These code samples show how to use SPLASH. To see projects utilizing SPLASH that you can run on your Event Stream Processor, refer to *Projects Utilizing SPLASH*.

Internal Pulsing

A stock market feed is a good example of several updates flowing into a stream.

Suppose the stock market feed keeps the last tick for each symbol. Some of the downstream calculations might be computationally expensive, and you might not need to recalculate on every change. You might want to recalculate only every second or every ten seconds. How can you collect and pulse the updates so that the expensive recalculations are done periodically instead of continuously?

The dictionary data structure and the timer facility allow you to code internal pulsing. Let's suppose that the stream to control is called InStream. First, define two local variables in the Flex operator:

```
integer version := 0;
dictionary(typeof(InStream), integer) versionMap;
```

These two variables keep a current version and a version number for each record. The SPLASH code handling events from the input stream is:

```
{
  versionMap[InStream] := version;
}
```

The special Timer block within the Flex operator sends the inserts and updates:

```
{
  for (k in versionMap) {
    if (version = versionMap[k])
      output setOpcode(k, upsert);
  }
  version++;
}
```

You can configure the interval between runs of the Timer block in numbers of seconds. Only those events with the current version get sent downstream, and the version number is incremented for the next set of updates.

This code works when InStream has only inserts and updates. It's a good exercise to extend this code to work with deletes.

Order Book

One example inspired by stock trading maintains the top of an order book.

Suppose there is a stream called Bid of bids of stocks (the example is kept simple by not considering the offer side), with records of the type:

```
[integer Id; | string Symbol; float Price; integer Shares; ]
```

where Id is the key field, the field that uniquely identifies a bid. Bids can be changed, so not only might the stream insert a new bid, but also update or delete a previous bid.

The goal is to output the top three highest bids any time a bid is inserted or changed for a particular stock. The type of the output where Position ranges from 1 to 3 is:

```
 [integer Position; | string Symbol; float Price; integer Shares; ]
```

For example, suppose the Bids have been:

```
[Id=1; | Symbol='IBM'; Price=43.11; Shares=1000; ]
[Id=2; | Symbol='IBM'; Price=43.17; Shares=900]
[Id=3; | Symbol='IBM'; Price=42.66; Shares=800]
[Id=4; | Symbol='IBM'; Price=45.81; Shares=50]
```

With the next event:

```
[Id=5; | Symbol='IBM'; Price=46.41; Shares=75]
```

The stream should output the records

```
[Position=1; Symbol='IBM'; | Price=46.41; Shares=75]
[Position=2; Symbol='IBM'; | Price=45.81; Shares=50]
[Position=3; Symbol='IBM'; | Price=43.17; Shares=900]
```

Note: The latest value appears at the top.

One way to solve this problem is with an event cache that groups by stock and orders the events by price:

```
eventCache(Bids[Symbol], coalesce, Price desc) previous;
```

The following code outputs the current block of the order book, down to the level specified by the `depth` variable.

```
{
  integer i := 0;
  string symbol := Bids.Symbol;
  while ((i < count(previous.Id)) and (i < depth) ) {
    output setOpcode([ Position=i; Symbol = symbol; |
                       Price=nth(i,previous.Price);
                       Shares=nth(i,previous.Shares);
                       ], upsert);
    i++;
  }
  while (i < depth) {
```

```
    output setOpcode([ Position=i; Symbol=symbol ], safedelete);
    i++;
  }
}
```

CHAPTER 8 Projects Using SPLASH

Two projects demonstrate how SPLASH is used.

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
;
```

Index

P

R

S

T

U

V