SYBASE®
An SAP® Company

SPLASH Tutorial

Sybase Aleri Streaming Platform 3.2

DOCUMENT ID: DC01288-01-0320-02

LAST REVISED: December, 2010

countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Group Ltd.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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

Table of Contents

About This Guide

1. Related Documents

This guide is part of a set. The following list briefly describes each document in the set.

Product Overview	Introduces the Aleri Streaming Platform and related Aleri products.
Getting Started - the Aleri Studio	Provides the necessary information to start using the Aleri Studio for defining data models.
Release Bulletin	Describes the features, known issues and limitations of the latest Aleri Streaming Platform release.
Installation Guide	Provides instructions for installing and configuring the Streaming Processor and Aleri Studio, which collectively are called the Aleri Streaming Platform.
Authoring Guide	Provides detailed information about creating a data model in the Aleri Studio. Since this is a comprehensive guide, you should read the *Introduction to Data Modeling and the Aleri Studio*. first.
Authoring Reference	Provides detailed information about creating a data model for the Aleri Streaming Platform.
Guide to Programming Interfaces	Provides instructions and reference information for developers who want to use Aleri programming interfaces to create their own applications to work with the Aleri Streaming Platform. These interfaces include: • the Publish/Subscribe (Pub/Sub) Application Programming Interface (API) for Java • the Pub/Sub API for C++ • the Pub/Sub API for .NET • a proprietary Command & Control interface • an on-demand SQL query interface
Utilities Guide	Collects usage information (similar to UNIX® man pages) for all Aleri Streaming Platform command line tools.
Administrators Guide	Provides instructions for specific administrative tasks related to the Aleri Streaming Platform.
Introduction to Data Modeling and the Aleri Studio	Walks you through the process of building and testing an Aleri data model using the Aleri Studio.
SPLASH Tutorial	Introduces the SPLASH programming language and illustrates its capabilities through a series of examples.
Frequently Asked Questions	Answers some frequently asked questions about the Aleri Streaming Platform.

Chapter 1. Introduction

The Sybase® Aleri Streaming Platform can be thought of as a programming language with two levels. The top level is a dataflow language of streams. It includes different stream primitives, for example, Join and Aggregate, and means of directing the data flow. Much of the work of building a data model is selecting the appropriate stream types, configuring them, and connecting the streams together.

The bottom level is an embedded language called SPLASH. SPLASH contains a simple language of expressions—as embedded in, say, Microsoft® Excel®—to compute values from other values. It also has variables, looping constructs, data structures, and helpful tools for debugging.

SPLASH's syntax is similar to C and Java®. It's spirit is close to little languages like AWK or Perl that solve relatively small programming problems. The SPLASH language is best described in little examples, which is what this tutorial does.

This tutorial is not exhaustive. Rather, it illustrates the kind of programming you can do in SPLASH. If you need more details on features such as built-in functions, refer to the *Authoring Reference*.

Chapter 2. Basics

2.1. First Program

Here's the SPLASH version of the classic “hello world” problem. The statement

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

is a simple bit of SPLASH code terminated by a semicolon. This statement prints the line `hello world` in the console of the Aleri Studio or on the terminal where the Sybase Aleri Streaming Processor was started.

2.2. Constants and Simple Expressions

The “hello world” code has a string constant in it. Note the use of single quotes around the string constant. That is different from C/C++ and Java, which use double quotes for string constants. Single quotes are a legacy of the syntax of SQL, which SPLASH follows to a large extent.

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 (1.065d) followed with a 'd' or 'D'. Expressions can be formed from the standard arithmetic operators (+, -, *, /, ^) and parentheses, with precedence defined as usual. For instance,

```
1 + 7 * 8
```

computes 57, whereas

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

computes 64. SPLASH includes a host of arithmetic functions too, for example,

```
sine(1.7855)
```

returns the sine of the value.

Like 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.

2.3. Null Values

Each data type 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. For example,

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

returns 3.

2.4. Variables and Assignment

Variables can be declared and assigned values in SPLASH. For instance,

```
int32 eventCount;
```

declares a variable named `eventCount` holding values of type "int32" (32-bit integers). You can assign the variable using the operator `:=`, such as

```
eventCount := 4;
```

and use the value of the variable by writing its name, such as

```
eventCount + 1
```

For brevity, you can mix declarations and assignments to initial values, such as

```
double 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. For example, the declaration

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

describes three variables, all of type double, with `pi` and `e` set to initial values.

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

```
int64 "500miles" := 500 * 1760;
string "for" := 'for ever';
```

This feature comes directly from SQL.

2.5. Types

Explicit variable declarations make SPLASH into a statically typed language like C and Java. That's 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.

SPLASH comes with a number of primitive data types. Besides int32, string, and double (for double-precision floating point numbers), SPLASH includes int64 (64-bit integers), money (for exact computations involving fixed-point decimal numbers), date (date/time values with second granularity, and timestamp (date/time values with millisecond granularity).

Programs in SPLASH automatically convert values of numeric type to other types if possible. For instance, if you declare

```
double e := 2.718281828459;
int32 years := 10;
```

then the expression

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

is a legal expression with type double. The variable `years` of type int32 is automatically converted to double, as is the constant 1000.0d of type money. The complete set of coercion rules can be found in the *Authoring Reference*. Downcasting (converting a number with more precision into a number with less precision) can be done using the `cast` operation. For instance, the expression

```
cast(int32, 1000.0d * (e ^ (0.05 * years)))
```

converts the double value into an int32 by truncating the decimal part of the number.

You can compute with values of type date and timestamp just as if they are numeric values. For instance, 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.

2.6. Type Abbreviations

You can give alternative names to types with the `typedef` declaration. For example,

```
typedef money euros;
```

declares `euros` to be another name for the money type, and

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

declares a variable `price` of that type. Type abbreviations are most useful when working with longer type names. Some more useful cases are given later in this tutorial.

You can also use the `typeof` operator to simplify type definitions. This operator returns the type of the expression. For instance,

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

is an equivalent way of writing

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

2.7. Blocks

Blocks of statements are written between curly braces. For example,

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

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.

Blocks can be nested, and the usual scoping rules apply. The SPLASH code

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

will print “there” instead of “here”.

2.8. Control Structures

The control structures of SPLASH are lifted directly from C and Java.

Conditional execution of statements is done with the `if` statement and the `switch` statement. For example, the block

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

sets a string variable to different values depending on the sign of the temperature. 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 int32, int64, money, double, date, timestamp, or string.

The `while` statement encodes loops. For example,

```
int32 i := 0, squares := 0;
while (i < 10) {
  squares := squares + (i * i);
  i++;
}
```

computes the sum of the squares of 0 through 9. 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 Sybase Aleri Streaming Platform, just the block.

Chapter 3. Record Events

In the Sybase Aleri Streaming Platform, streams process “record events.” An event is a record, which is a composite value that associates field names to values, and an operation (for example, insert or update). Some of the fields are designated as key fields so that the operation can be applied to existing records.

3.1. Record Types

Here's an example of a record type:

```
[ string Symbol; | int32 Shares; double Price; ]
```

The type describes a record with three fields: a string field called Symbol, which is the sole key field because it's to the left of the `|` symbol; an int32 field representing the number of shares; and a double field representing the price.

Each field must have one of the basic types (int32, int64, double, money, string, date, or timestamp). Records cannot be nested.

Because record types are long, it's often helpful to use `typedef` to give them a shorter name, as in

```
typedef [ string Symbol; | int32 Shares; double Price; ] rec_t;
```

The examples below use this shorter name.

3.2. Record Values

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, for example,

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

declares a record variable and assigns a record value to it.

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. For example,

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

changes the value of the Shares field to 80.

3.3. 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 with 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; ]
```

A third record

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

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

3.4. Record Casting

Records are implicitly coerced depending on their context. Extra fields are dropped and missing fields are made null. For instance, the assignment

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

drops the Extra field before the assignment is made. Conversely,

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

assigns the variable `rec` to a record whose Price field is null.

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

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

it will make the Symbol field into a key field. Key fields should not be null, however. It's legal to assign

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

but you cannot send this to downstream streams. This will be described in more detail below.

3.5. Hidden Fields

Records also have two distinguished fields, `rowid` and `rowtime`, of type int64 and date respectively. Streams fill in these values automatically, and you can access them with the usual "." operation.

3.6. Operations

Events have one more piece of data implicit within them, namely 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.

You 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,

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

```
setOpcode(v,safedelete);
print('opcode=', string(getOpcode(v)), '\n');
```

prints the numeric codes for insert (which is 1) and safedelete (which is 13).

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

Chapter 4. Functions

4.1. SPLASH Functions

SPLASH contains a large number of built-in functions; see *Authoring Reference* for the complete list. 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. 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) { ... }
```

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:

```
int32 factorial(int32 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(int32 x) {
   if (x = 1) {
      return 'odd';
   } else {
      return even(x-1);
   }
 }
string even(int32 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.

```
int32 sumFun(int32 x, int32 y) {
  return x+y;
}
string getField([ int32 k; | string data;] rec) {
  return rec.data;
}
```

The real use of SPLASH functions is to define and debug 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

```
double bondValue(double currentPrice,
                 int32 daysToMature,
                 double inflation)
{
  ...
}
```

and use it in many places within the data model.

4.2. C and Java Functions

You can also write your own functions in C/C++ or Java. That's an advanced feature, and the *Authoring Reference* gives the recipe for how to build libraries and call them from within the Sybase Aleri Streaming Platform.

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.

5.1. 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(double) roots;
int32 i := 0;
double 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, as in

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

The old vector is automatically thrown away (garbage collected in the parlance of programming languages).

5.2. Dictionaries

A dictionary associates keys to 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 the statement

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

overwrites the previous value associated with the key “EUR”.

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,

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

creates an empty dictionary and assigns it to `convertFromUSD`.

Section 7.1, “Internal Pulsing” gives an example using a dictionary.

5.3. Mixing Vectors and Dictionaries, and Reference Semantics

The previous examples of vectors and dictionaries store simple data types. There is nothing, however, that prevents you from building 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. The declaration and function

```
dictionary(string, vector(money)) previousPrices;
int32 addPrice(string symbol, money price)
{
  vector(money) prices := previousPrices[symbol];
  if (isnull(prices)) {
    prices := new vector(money);
    previousPrices[symbol] := prices;
  }
  push_back(prices, price);
}
```

create such a set of stored prices, keyed by symbol.

The example relies on reference semantics of containers. For instance, the assignment

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

returns a reference to the vector, not a copy of the vector. 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,

```
dictionary(int32, int32) d0 := new dictionary(int32, int32);
dictionary(int32, int32) d1 := d0;
d1[0] := 1;
if (d0[0] = 1) print('aliased!');
```

results in the program printing “aliased!”.

5.4. 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, the declaration

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

declares an event cache for the input stream Trades. You can have as many event caches as you wish per input stream, so you can declare

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

in the same stream.

By default, the buckets are determined by the keys of the input stream. For example, if you have an input stream Trades with

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

there are two buckets, one for events with Symbol “T” and one with symbol “CSCO”. Each event—whether an insert, update, or delete—is put into the corresponding bucket. For instance, if a delete event carrying

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

comes into the stream, there are two events in the bucket for “CSCO”. You can change that behavior by declaring the event cache to coalesce events, as in

```
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, you write `sum(events.Shares)`. Which bucket gets selected? By default, it's the bucket with associated with the current event that came from the input stream. You can change the bucket with the `keyCache` operation, described more fully in the *Authoring Reference*.

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 the same number of shares and the same symbol

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

or in one great big bucket

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

You can also order the events in the bucket by field. For example,

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

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 with

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

or a maximum age of 20 seconds with

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

or both

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

The *Authoring Reference* gives more detail on this option. It also describes the `expireCache` function for manually expiring events from a bucket.

Section 7.2, “Order Book” gives an example using an event cache.

Chapter 6. Integration with FlexStreams

FlexStreams use 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.

6.1. Access to the Event

When an event arrives to a FlexStream 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 Input-Stream, the variables are

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

When the method for InputStream 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.

(An aside: delete events always come populated with the data previously held in the input stream.)

A FlexStream 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.

6.2. Access to Input Streams

Within method and timer code in FlexStreams, you can also 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 other variable is for advanced users. It is described in more detail in the *Authoring Reference*.

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

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

You might have another input stream 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 with

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

This statement finds 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.

Conversely, when processing events from the Trades stream, you can look up earnings data with

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

The syntax here uses curly braces rather than square brackets because the meaning is different. The Trades event doesn't 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 mean 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 as

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

which 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.

In a FlexStream, you can access records in the stream itself. For instance, if the FlexStream is called Flex1, you can write a loop

```
for (rec in Flex1_stream) {
  ...
}
```

just as you can with any of the input streams.

6.3. Output Statement

Typically, a FlexStream 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, you use the `output` statement.

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

```
int32 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.

6.4. Notes on Transactions

Each method in FlexStream methods processes one event at a time. The Sybase Aleri Streaming Plat-

form can, however, be fed data in transaction blocks---that is, groups of insert, update, and delete events. How are these processed?

The short answer is that the method is run on each event in the transaction block. The full answer is for the advanced or curious user.

The Sybase Aleri Streaming Platform maintains an invariant: a stream takes in a transaction block, and produces a transaction block. It's always one block in, one block out. A FlexStream 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 FlexStream 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 will be rejected. For example, if you try to output a record with any null key columns, like

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

the whole transaction block will be rejected. Likewise, if you try

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

(which is implicitly an insert), and there is already a record in the FlexStream with Id set to 4, the block will be rejected. You can get a report of bad transaction blocks by starting the Sybase Aleri Streaming Platform 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 will be accepted.

A final note: 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 will be sent downstream. If your code outputs an insert followed by a delete, both events will be removed from the transaction block. Thus, you might output many events, but the transaction block might contain only some of them.

Chapter 7. Examples

Here are a few examples for putting SPLASH to use.

7.1. Internal Pulsing

Suppose you have a lot of updates flowing into a stream. A good example is a stock market feed that keeps the last tick for each symbol. Some of the downstream calculations might, however, be computationally expensive, and you might not even 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, you define two local variables in the FlexStream:

```
int32 version := 0;
dictionary(typeof(InStream), int32) 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 FlexStream 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.) Notice how only those events with the current version get sent downstream, and how the version number gets 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.

7.2. 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

```
[int32 Id; | string Symbol; double Price; int32 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 is

```
 [int32 Position; | string Symbol; double Price; int32 Shares; ]
```

where Position ranges from 1 to 3.

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]
```

and the next event is

```
[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 how 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 depth specified by the variable `depth`.

```
{
  int32 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++;
  }
}
```