Apama EPL Streams

A Short Tour

January 2014

Get There Faster

Table of contents

Introduction

Apama EPL allows code authors to express event-driven programs using natural event-processing constructs. An EPL program consists of a set of interacting monitors that receive, process and emit events. Monitor instances are self-contained, communicating with other monitor instances via events. An Apama application can thus be viewed as a dynamic network of interacting monitor instances communicating via events. Why dynamic? Because the application creates and destroys monitor instances in response to the external events received; similarly, the monitor instances dynamically subscribe and unsubscribe to particular event patterns or complex event expressions as needed. Thus, at any given instant, the application has only the monitor instances it needs and is only listening for the events of interest at that time. This novel approach makes Apama a highly efficient and responsive tool for complex event processing.

Complex event processing systems come in different flavors, one of which is *event stream processing*. The event stream processing approach is similar to the Apama approach, but tends to involve networks that are much less dynamic. These networks are constructed from *streams* and *processing nodes*, where a processing node is typically a *query*, defined using declarative, relational language elements.

Event stream processing is useful in cases where one or more flows of raw events are to be converted into a set of 'refined' flows of added-value events. For these operations, the use of event stream processing language elements allows these operations to be expressed more clearly and concisely than when using procedural language constructs. For this reason, Apama EPL includes event stream processing elements.

The event stream processing constructs in EPL maintain the Apama ethos of *operational responsiveness*. Thus you will find that Apama stream queries are not static and that they are closely integrated with the rest of the EPL language. Application developers can write code to add and remove stream queries as required, and the *streams* language elements allow the values controlling the stream query behavior to be varied dynamically.

The Apama event stream processing model

The Apama event stream processing model consists of a network of streams and processing nodes; a processing node whose logic is expressed in terms of a relational query expression is a stream query.

The diagram below shows an example of a stream processing network.

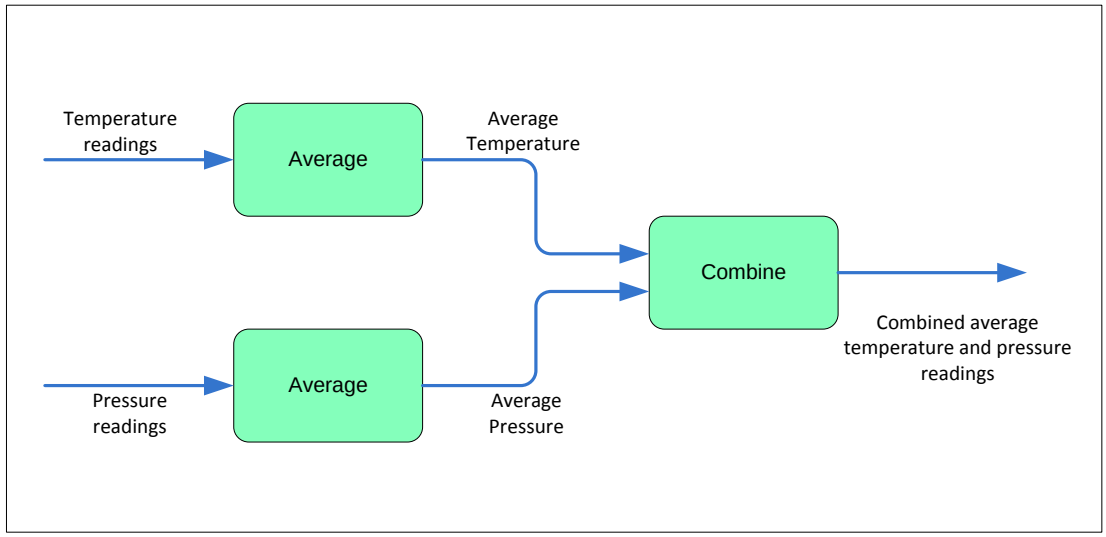

FIGURE 1: STREAM PROCESSING NETWORK

The network consists of five *streams*[1] and three *stream queries*. Each stream query has one or more input streams, from which it receives events, and one output stream, to which it transmits events.

In Apama, each event stream has a single *generator* but can have multiple *consumers*. Each stream or stream query is created within and owned by an Apama monitor instance. The streams and stream queries within a monitor instance are used to convert the events received by the monitor instance into added-value events. These added-value events are then available for use by standard EPL *actions*.

Example events

Most of the examples in this tutorial use the following events

```
event Temperature {
  string sensorId;
  float temperature;
}
```

```
event Pressure {
  string sensorId;
  float pressure;
}
```

```
event TemperatureAndPressure {
  string sensorId;
  float temperature;
  float pressure;
}
```

[1] *In Apama, the term 'stream' is used to refer both to the channel through which the events flow and also to the events flowing through the channel. Some members of the CEP fraternity use the term event channel to refer to the former and event stream to refer to the latter. In Apama, the term channel is already in use and so stream is used to refer to the 'event channels' connecting stream queries.*

C1

Processing events using streams

To receive events directly into a listener action, an `on` statement is used - for example:

```
01. Temperature t;
02. on all Temperature(sensorId="S001"):t { print t.toString(); }
```

C2

If, instead, the events are to be received into a stream, a stream assignment statement is used:

```
01.   stream<Temperature> temperatures := all Temperature(sensorId="S001");
```

C3

This statement declares the stream variable `temperatures`, which is used to refer to a stream of `Temperature` events. On the right side of the assignment, the `'all Temperature(sensorId="S001")'` expression is a stream source template. A stream source template is an event template preceded by the `all` keyword; it uses no other event operators. It creates a stream that contains events that are received by the monitor instance and that match the event template.

```
01.    Temperature temperature;
02.    stream<Temperature> temperatures := all Temperature(sensorId="S001");
03.    from t in temperatures retain 3
04.    select Temperature("S001", mean(t.temperature)) : temperature {
05.      print temperature.toString();
06.    }
```

C4

Let's look in detail at the `from` statement. A `from` statement is similar to an `on` statement in form. It consists of three parts:

(a) a stream query

```
from t in temperatures retain 3 select Temperature("S001", mean(t.temperature))
```

(b) followed by a co-assignment

```
: temperature
```

(c) followed by a listener action

```
{ print temperature.toString(); }
```


In this example, the stream query processes events from the `temperatures` stream and computes the average temperature value of the three *most recent* events. A new output event is created for each new input event, having the literal value `"S001"` for the sensorId field and the evaluated average temperature value for the temperature field. Each output event, in turn, is co-assigned to the variable `temperature` and this is used in the `print` statement, within the listener action.

The average temperature value is calculated using the built-in[2] `mean()` aggregate function.

Below, we give a 'whirlwind tour' of the streams language elements.

CREATING A STREAM NETWORK

The code example below implements the simple stream network illustrated in Figure 1. The code illustrates that stream queries can be used (a) in `from` statements and also (b) on the right side of a stream assignment.

Executing a stream assignment statement does two things. It creates, within the stream network, the defined query. It then updates the *stream variable* (on the LHS of the assignment) to refer to the query's output stream.

Up to now we have referred to *streams* as 'event streams'. In Apama, the type of a stream need not be an event; it is possible to create streams of simple types such as decimal, float, integer, boolean, string.[3]

```
01. TemperatureAndPressure tp;
02. stream<Temperature> temperatures := all Temperature(sensorId="T001");
03. stream<Pressure> pressures := all Pressure(sensorId="P001");
04. stream<float> meanTs := from t in temperatures retain 3 select mean(t.temperature);
05. stream<float> meanPs := from p in pressures retain 3 select mean(p.pressure);
06. from t in meanTs retain 1 from p in meanPs retain 1
07. select TemperatureAndPressure("S001",t,p) : tp {
08.   print tp.toString();
09. }
```

C5

Line 6 of the code example shows one method for joining two streams. The stream query contains two `from` clauses, where each `from` clause specifies that the most recent item in the stream is retained. A query with two `from` clauses identifies that a *cross-join* operation should be performed between the two source item sets. In the code example, when a new item is available on the `meanPs` stream, it is joined with the most recent item on the `meanTs` stream, and when a new item is available on the `meanTs` stream, it is joined with the most recent item on the `meanPs` stream.

[2] *Apama provides a number of commonly used aggregates as predefined 'built-in' aggregates. It is also possible to create user-define 'custom' aggregates.*
[3] *It is for this reason that, in the 'Developing Apama Applications in EPL' manual, and in other documentation, we refer to the contents of streams as 'items', not as 'events'.*

USING INLINE 'STREAM SOURCE TEMPLATE' EXPRESSIONS

The code example above can be re-written in a more concise format by writing the stream source template expressions inline, as illustrated below.

```
01. TemperatureAndPressure tp;
02. stream<float> meanTs :=
03.   from t in all Temperature(sensorId="T001") retain 3 select mean(t.temperature);
05. stream<float> meanPs :=
06.   from p in all Pressure(sensorId="P001") retain 3 select mean(p.pressure);
07. from t in meanTs retain 1 from p in meanPs retain 1
08. select TemperatureAndPressure("S001",t,p) : tp {
09.   print tp.toString();
10. }
```

C6

USING COMPOUND QUERIES

The complete stream network for this example can be expressed as a single compound query.

```
01. TemperatureAndPressure tp;
02. from t in
03.   from t in all Temperature(sensorId="T001") retain 3 select mean(t.temperature)
04.   retain 1
05. from p in
06.   from p in all Pressure(sensorId="P001") retain 3 select mean(p.pressure)
07.   retain 1
08  select TemperatureAndPressure("S001",t,p) : tp {
09.   print tp.toString();
10. }
```

C7

Note that the *item identifiers*, `t` and `p`, in the `from` clauses for the inner queries use the same names as those in the outer queries. This does not cause any ambiguity because the scope of the item identifier in the inner query is restricted to the inner query, and within the inner query hides the name used in the outer query. Hence, the item identifier, `t`, in the inner query refers to `Temperature` events from the stream `all Temperature(sensorId="T001")`, whereas the item identifier, `t`, in the outer query refers to the `float` items produced by the inner query. Using the same identifier is a matter of style; different identifiers could be used if preferred (for example, `avgT` and `t`).

USING DYNAMIC VALUES IN STREAM QUERIES

One of the great features of Apama stream queries is that the values used in the stream query expression can be dynamically changed throughout the lifetime of the query. This is useful (for example) for setting dynamic thresholds or for changing the aggregation period of a query. The code examples below illustrate these cases.

```
01. TemperatureAlert alert;
02. from t in all Temperature(sensorId="T001") where t.temperature > threshold
03. select TemperatureAlert(t.sensorId,t.temperature): alert { emit alert; }
```

C8

```
01. TemperatureRange range;
02. from t in all Temperature(sensorId="T001") within period every period
03. select TemperatureRange(t.sensorId,min(t.temperature),max(t.temperature)): range {
04.   print range.toString();
05. }
```

C9

In the code examples above, if the variables `threshold` and `period` are local variables[4] , then the value used by the queries are the values of the local variables has when the `from` statement is executed.[5] Even if the local variable is assigned a new value at some later point in the program execution, the values used by the queries will be constant throughout the lifetime of the query.

However, if global variables[6] or event member variables[7] are used and, at a later time, the values of these variables are changed, then these value changes *will* affect the behavior of the stream queries. The full code examples for the dynamic use-cases are given below.

```
01. event Temperature { string sensorId; float temperature; }
02. event TemperatureAlert { string sensorId; float temperature; }
03. event ChangeThreshold { float temperature; }
```

```
01. monitor TemperatureAlertMonitor {
02.   float threshold := 60.0; // a global variable is used
03.   action onload() {
04.     TemperatureAlert alert;
05.     from t in all Temperature(sensorId="T001") where t.temperature > threshold
06.     select TemperatureAlert(t.sensorId,t.temperature): alert { emit alert; }
07.     ChangeThreshold ct;
08.     on all ChangeThreshold():ct { threshold := ct.temperature; }
09.   }
10. }
```

[4] *A local variable is defined within the body of an action.*
[5] *This is exactly the same mechanism as is used when creating event listeners (that is, when using `on` statements).*
[6] *When the stream query is defined within a monitor action.*
[7] *When the stream query is defined within an event action.*

C10

```
01. event Temperature { string sensorId; float temperature; }
02. event TemperatureRange { string sensorId; float minTemperature; float maxTemperature; }
03. event ChangePeriod { float period; }
```

```
01. using com.apama.aggregates.max; using com.apama.aggregates.min;
02. event TemperatureRangeService {
03.   float period;  // an event member variable is used
04.   action init( string id, float _period ) {
05.     period := _period;
06.     TemperatureRange range;
07.     from t in all Temperature(sensorId=id) within period every period
08.     select TemperatureRange(id,min(t.temperature),max(t.temperature)): range {
09.       print range.toString();
10.     }
11.   }
12.   action setPeriod(float  period ) { period :=  period; }
13. }
14. monitor UsesTemperatureRangeService {
15.   action onload() {
16.     TemperatureRangeService trs := new TemperatureRangeService;
17.     trs.init("S001",60.0);
18.     ChangePeriod cp;
19.     on all ChangePeriod ():cp { trs.setPeriod(cp.period); }
20.   }
21. }
```

C11

USING STREAM VARIABLES

Because streams are values in EPL, we can pass stream references between the code elements within a monitor. This is useful when writing services. A common service (that is, a service used by two or more monitors) is normally implemented using a 'service event'. This event contains the logic to implement the service or to access an external service. A stream can be used as part of the interface to the service: the stream and stream query specification is encapsulated within the *service event* code and a reference to the stream created by this code is returned, from the service action to the client monitor code, as the return value of an action call. This is illustrated in the following code example.

```
01. event Temperature { string sensorId; float temperature; }
02. event TemperatureRange { string sensorId; float minTemperature; float maxTemperature; }
```

```
01. using com.apama.aggregates.max; using com.apama.aggregates.min;
02. event TemperatureRangeService {
03.   float period;
04.   action init( string id, float  period ) returns stream<TemperatureRange> {
05.     period :=  period;
06.     return
07.       from t in all Temperature(sensorId=id) within period every period
08.       select TemperatureRange(id,min(t.temperature),max(t.temperature));
09.   }
10. }
11. monitor UsesTemperatureRangeService {
12.   action onload() {
13.     TemperatureRangeService service := new TemperatureRangeService;
14.     stream<TemperatureRange> ranges := service.init("S001",60.0);
15.     TemperatureRange range;
16.     from r in ranges select r : range { print range.toString(); }
17.   }
```



```
18. }
```

C12

USING THE SHORT-FORM FROM STATEMENT

In example C12 on line 16 of the code, the query used is very simple. It merely selects the current item in the stream and co-assigns it to the variable `range`. This is a common use-case and the EPL language provides an alternate, short-form version that can be used instead, as illustrated below.

```
16.     from ranges: range { print range.toString(); }
```

C13

To further simplify the code in example C12, note that, instead of declaring a `ranges` stream variable, we can place the expression for the stream (that is, `"service.init("S001",60.0)"`) directly in-line, in the `from` statement:

```
16.     from service.init("S001",60.0): range { print range.toString(); }
```

C14

Hence, the monitor code in example C12 can be rewritten as

```
12. monitor UsesTemperatureRangeService {
13.   action onload() {
14.     TemperatureRangeService service := new TemperatureRangeService;
15.     TemperatureRange range;
16.     from service.init("S001",60.0): range { print range.toString(); }
17.   }
18. }
```

C15

STREAM LIFETIME

When considering the lifecycle of a stream, firstly we reflect on how they are created. A `from` statement is similar to an `on` statement, in that both create stream listeners. When creating the stream listener, a *listener variable* can be assigned to refer to the stream listener. The listener variable can then be used (at a later time) to *quit* the stream listener.[8]

When creating a stream query and assigning it to a *stream variable*, the stream variable can be used (at a later time) to *quit* the stream query.

Once created, a stream (and the stream query supplying it) remains in existence until any of the following occur:

(a) it is *quit*,

[8] *Nb. This is identical to an EPL on statement, where a listener variable can be used to quit a standard event listener.*

(b) all of its downstream connections are removed,
(c) the removal of an upstream stream means that the stream (stream query) can generate no more output.

The above statements sound rather complicated but are quite straightforward. Consider the following code example:

```
01. event Temperature { string sensorId; float temperature; }
02. event Quit { string what; }
```

```
01. using com.apama.aggregates.mean;
02. monitor StreamLifetimes {
03.   action onload() {
04.     float temperature;
05.     stream<Temperature> temperatures := all Temperature(sensorId="S001");
06.     stream<float> meanTs := from t in temperatures within 60.0
07.       select mean(t.temperature);
08.     listener freezing := from t in meanTs where t < 0.0 select t: temperature {
09.       print "It's freezing! The temperature is " + temperature.toString();
10.     }
11.     listener boiling := from t in meanTs where t > 100.0 select t: temperature {
12.       print "It's boiling! The temperature is " + temperature.toString();
13.     }
14.     on Quit("temperatures") { temperatures.quit(); }
15.     on Quit("meanTs")       { meanTs.quit(); }
16.     on Quit("freezing")     { freezing.quit(); }
17.     on Quit("boiling")      { boiling.quit(); }
18.   }
19. }
```

C16

In this example, the stream network consists of two streams (declared on lines 5 and 6-7) and two stream listeners (declared on lines 8-10 and 11-13). The stream variables `temperatures` and `meanTs` refer to the two streams, and the listener variables `freezing` and `boiling` refer to the two stream listeners. Let's take a look at what happens when `quit()` is called on each of the listener and stream variables:

- If `freezing.quit()` is called, then only the stream listener referred to by `freezing` becomes inactive. Similarly, if `boiling.quit()` is called, then only the stream listener referred to by `boiling` becomes inactive.
- If `meanTs.quit()` is called, then all of the streams, stream queries and stream listeners will become inactive. This is because the `meanTs` query is the only downstream connection for the `temperatures` stream, and once `meanTs` is quit, the two stream listeners for `freezing` and `boiling` can no longer produce any output.
- Finally, if `temperatures.quit()` is called, then there would be no further input to the stream query for `meanTs`. However, items in the window of the stream query may remain 'within the window' for up to 60.0 seconds after the `temperatures` stream is quit. Hence the `meanTs` stream query, and any queries/listeners downstream of it, will remain active until all items in the `meanTs` stream query window have 'expired'.

USING WINDOWS

Various examples in earlier sections have used *window* operators. Within a stream query, when a window operator is applied to a stream, it causes some of the past items in the stream to be retained. These are the items upon which the relational query operations are performed. For example, the query

```
from t in all Temperature(sensorId="T001") retain 10 select mean(t.temperature)
```

C17

will calculate, for sensor `"T001"`, the mean temperature value from the set of the most recent 10 temperature readings from that sensor, and the query

```
from t in all Temperature(sensorId="T001") within 60.0 select mean(t.temperature)
```

C18

will calculate, for sensor `"T001"`, the mean temperature value from the set of all temperature readings for that sensor within the last 60.0 seconds.

The table below gives a guide to the window operators and their combinations:

`retain all`	Retains all of the items input to the stream since its creation[9].
`retain` *`number`*	Retains (up to) the *`number`* most recent items input to the stream
`within` *`duration`*	Retains all items input to the stream within the last *`duration`* seconds.
`within` *`duration`* `retain` *`number`*	Retains (up to) the *`number`* most recent items input to the stream within the last *`duration`* seconds.
`retain` *`number`* `with unique` *`key`*	Retains (up to) the *`n`* most recent items input to the stream. A new item with a given *`key`* value will displace an existing item with the same `key` value.
`within` *`duration`* `with unique` *`key`*	Retains items input to the stream within the last *`duration`* seconds. A new item with a given *`key`* value will displace an existing item with the same *`key`* value.

If no window operator is applied to a stream then the set of items on which the relational query operations are performed is the set of items that is *current* for the stream. Using a stream without applying any window operations to it can be useful when used within a join query.

[9] *Note. The implementation achieves this behavior without actually retaining all of the items.*

USING JOINS

There are two types of joins that can be used within a stream query: cross-joins and equijoins.

A cross-join of two sets combines every item from one set with each item from the other set. A cross-join is performed by using two, top-level *from* clauses in a query. We have already seen an example of this:

```
01. TemperatureAndPressure tp;
02. stream<Temperature> temperatures := all Temperature(sensorId="T001");
03. stream<Pressure> pressures := all Pressure(sensorId="P001");
04. stream<float> meanTs := from t in temperature retain 3 select mean(t.temperature);
05. stream<float> meanPs := from p in pressure retain 3 select mean(p.pressure);
06. from t in meanTs retain 1
07. from p in meanPs retain 1
08. select TemperatureAndPressure("S001",t,p) : tp {
09.   print tp.toString();
10. }
```

C19

An equi-join is performed by following the initial *from* clause with a *join* clause. An equi-join of two sets combines items in the two sets where a specified *key value* of the item in the first set matches a specified *key value* of the item in the second set. Separate *key value expressions* for each source item identify the key values to be compared.

```
01. TemperatureAndPressure tp;
02. from t in all Temperature() partition by t.sensorId retain 1
03. join p in all Pressure() partition by p.sensorId retain 1
04  on sensorNumber(t.sensorId) equals sensorNumber(p.sensorId)
05. select TemperatureAndPressure(combinedId(t.sensorId), t.temperature, p.pressure) : tp {
06.   print tp.toString();
07. }
```

C20

When considering performance, cross-joins will in general be less efficient than equijoins. It is advised that cross-joins only be used where the number of items in the stream windows is small (as in code example C19).

Note that joins can be performed between a stream[10] and a window. For example:

```
01. TemperatureAndPressure tp;
02. stream<Temperature> temperatures := all Temperature(sensorId="T001"); 1
03. stream<Pressure> pressures := all Pressure(sensorId="P001");
04  from t in temperatures from p in pressures retain 1
05.   select TemperatureAndPressure ("S001",t.temperature,p.pressure) : tp {
06.   print tp.toString();
07. }
```

[10] *That is, where no window operators are applied to the stream, in the query.*

C21

This join will produce an output item whenever there is a new `Temperature` event for the sensor but not when there is a new `Pressure` event. The temperature and pressure events arrive at different times; when the temperature event arrives, because of the `retain 1` in the right side `from` clause, there is a pressure event available for joining with; but, because there is no window operation in the left side `from` clause, when a pressure event arrives, there is no temperature event to join with.

USING PARTITIONS AND GROUPS

Code example C20 used the `partition by` clause. The `partition by` clause splits a stream into partitions, based on a key value. When a window operator is applied to a partitioned stream, the behavior is as if a separate window operator had been applied to each partition. We often refer to the result of using `partition by` followed by a window operator as a partitioned window; queries with partitioned windows are used to retain a set of items for each partition, as illustrated in the earlier code example, C20. In this example

```
01. Temperature temperature;
02. from t in all Temperature() partition by t.sensorId retain 3
03. group by t.sensorId select Temperature(t.sensorId, mean(t.temperature)):
04.   temperature {
05.   print temperature.toString();
06. }
```

C22

The combined `partition by` and `retain` clauses cause the last three values for each sensor to be retained. By contrast, the `group by` clause's effect is to alter the behavior of the projection (the item generated by the `select` clause) such that aggregate values are generated for each group in the collection and not for the collection as a whole. For example, when a new `Temperature` event occurs for sensor `"S001"`, the event will be directed to the partition for that sensor. It will cause the window contents for that partition to change, which, in turn, will affect the collection of events over which the aggregate projection is being performed; because a `group by` clause is present, a new projected value will be produced only for the group(s) affected by the update (in this case, the group for sensorId `"S001"`) . So, the end result is that an incoming temperature event, for sensor `"S001"`, causes a new outgoing mean temperature event for sensor `"S001"` to be produced. The `group by` clause can also be used without `partition by`, as in the following code sample.[11]

[11] *Note that (as implied by the example), there is usually little point in partitioning a time-based (a `within`) window. One exception to this is when it is combined with the `with unique` clause.*


```
01. Temperature temperature;
02. from t in all Temperature() within 60.0
03. group by t.sensorId select Temperature(t.sensorId, mean(t.temperature)):
04.   temperature {
05.     print temperature.toString();
06.   }
```

C23

USING RSTREAM

Normally, in stream queries, we select items that are currently *in* the stream or window. Adding the keyword `rstream` to a `select` clause causes it to select the items that are currently leaving the stream or window. The main use of this is to delay events, either by a time period or by a number of events. The delayed event is typically compared to the set of events that arrived after it, up until the current time, as illustrated by the code example below.

```
01. stream<float> tNow := from t in all Temperature(sensorId="T001")
02.   select t.temperature;
03. stream<float> tDelayed := from t in tNow retain 10 select rstream t;
04. float t;
05. from t1 in tDelayed from t2 in tNow retain 10 where t2 > t1 * 1.05 select t2 : t
06.   print "Rapid temperature rise: " + t.toString();
07. }
```

C24

Common Patterns

This section lists a few common patterns. You have seen many of them in the earlier code examples.

AGGREGATION

We have seen examples of this already, calculating the 'running averages' of the temperature and pressure readings. A common use-case, illustrated below, is the calculation of the volume-weighted average price of a stock. This example used the weighted-average aggregate function, `wavg()`.

```
01. using com.apama.aggregates.wavg;
02. event Tick { string symbol; decimal price; decimal volume; }
03. monitor CalculateVwap {
04.   action onload() {
05.     decimal vwap;
06.     from t in all Tick(symbol="SOW") within 300.0 select wavg(t.price,t.volume): vwap {
07.       print vwap.toString();
08.     }
09.   }
10. }
```


C25

Aggregation can also be used in combination with `group by` to generate the aggregate results for different groups of items, as illustrated in code examples C22 and C23. Note that code authors are not restricted to the set of built-in aggregates as it is possible to define *custom* aggregates.

THROTTLING

Sometimes it is the case that results are only required at a given rate. We can extend the example above, adding an `every` clause, so that the query only generates values every 10 seconds.

```
01. using com.apama.aggregates.wavg;
02. event Tick { string symbol; float price; float volume; }
03. monitor CalculateVwap {
04.   action onload() {
05.     float vwap;
06.     from t in all Tick(symbol="SOW") within 300.0 every 10.0
07.     select wavg(t.price,t.volume): vwap {
08.       print vwap.toString();
09.     }
10.   }
11. }
```

C26

DYNAMIC FILTERS

Event listeners, created using `on` statements, are very efficient at matching events, but have the drawback that the values of any variables or expressions used within an *event template* are evaluated only when the `on` statement is executed (that is, when the event listener is created) and remain fixed thereafter.

If we are using event listeners only and have a use-case where we need to change one of the match values, then, each time that the desired match value changes, we would need (a) to quit the current listener and (b) recreate it with the new match value. An alternative approach is to use streams. For example, if we want to receive `Temperature` events for a given sensor, but to select only those where the `temperature` value is greater than a given, static threshold, we would use

```
01. event Temperature { string sensorId; float temperature; }
02. monitor StaticFilter {
03.   action onload() {
04.     Temperature temperature;
05.     on all Temperature (sensorId="T001", temperature>38.0): temperature {
06.       print temperature.toString();
07.     }
08.   }
09. }
```

C27

If, instead, we need to change the temperature threshold dynamically, then the following code could be used:

```
01. event Temperature { string sensorId; float temperature; }
02. event Threshold { string sensorId; float temperature; }
03. monitor StaticFilter {
04.   Threshold threshold := Threshold("T001",38.0);
05.   action onload() {
06.     Temperature temperature;
07.     from t in all Temperature(sensorId="T001")
08.     where t.temperature > threshold.temperature select t : temperature {
09.       print temperature.toString();
10.     }
11.     on all Threshold(sensorId="T001"): threshold {}
12.   }
13. }
```

C28

In the static case (that is, where the threshold value does not change), the code in example C27 is more efficient than that of example C28, because the events that are not of interest are rejected as early as possible (that is, before being passed to the monitor instance). In the dynamic case (that is, where a changing threshold value is required), the C28 code example is more elegant and typically more efficient than using a non-streams approach.

In the dynamic threshold use-case, choosing which solution to prefer – using only event listeners or using streams - would depend on how frequently the threshold value is expected to change. The cost of quitting the current listener and recreating it with the new threshold value may be acceptable if the threshold value changes only infrequently.

JOINING THE MOST RECENT EVENT ON EACH OF TWO STREAMS

Another common pattern that has already been seen is that of comparing the most recent values from two event streams. The following code example illustrates this pattern with a use-case example of calculating the price spread between two stocks.

```
01. event Price { string symbol; float price; }
02. monitor ComputeSpreads {
03.   action onload() {
04.     float spread;
05.     from a in all Price(symbol="IBM") retain 1
06.     from b in all Price(symbol="MSFT") retain 1
07.     select a.price - b.price : spread {
08.       print spread.toString();
09.     }
10.   }
11. }
```

C29

RETAINING THE MOST RECENT ITEM IN EACH PARTITION OF A PARTITIONED STREAM

There are some situations where you want to join the most recent events from two sources, based on a common key. Typically you are processing all events from those sources and not a subset of those events. This pattern is similar to the previous example, but with a `partition by` clause added to each 'leg' of the join.

```
01. event Temperature { string sensorId; float temperature; }
02. event Pressure { string sensorId; float pressure; }
03. event TemperatureAndPressure { string sensorId; float temperature; float pressure; }
04. monitor CombineTheLatestTemperatureAndPressureReadings {
05.   action onload() {
06.     TemperatureAndPressure tp;
07.     from t in all Temperature() partition by t.sensorId retain 1
08.     join p in all Pressure() partition by p.sensorId retain 1
09.     on t.sensorId equals p.sensorId
10.     select TemperatureAndPressure(t.sensorId, t.temperature, p.pressure) : tp {
11.       print tp.toString();
12.     }
13.   }
14. }
```

C30

JOINING AN EVENT WITH A PREVIOUS EVENT

Another use-case that is reasonably common is where an item output from a stream query needs to be compared to the previous output item. As an example, let's say that we need to detect for a given sensor when the average temperature value *`was below`* a threshold value but now *is above* the threshold value.

```
01. using com.apama.aggregates.mean;
02. event Temperature { string sensorId; float temperature; }
03. monitor DetectBreach {
04.   action onload() {
05.     stream<float> temperatures := all Temperature(sensorId="S001");
06.     stream<boolean> current := from t in temperatures within 60.0
07.     select mean(t.temperatures) > 97.0;
08.     stream<boolean> previous := from c in current retain 1 select rstream c;
09.     string text;
10.     from c in current from p in previous where c and not p
11.     select "Temperature breach" : text {
12.       print text;
13.     }
14.   }
15. }
```

C31

Further reading

For more details on using Apama stream queries please refer to the Apama documentation. See the section “Working with Streams and Stream Queries” within “Developing Apama Applications in EPL”. You may also refer to the “Apama EPL Reference”.

