RedBorder CEP documentation

redborder

Version 1.0.0

Índice

Capítulo 1: Introducción a redBorder CEP

En este capítulo daremos una breve introducción a redBorder CEP y a su arquitectura.

RedBorder CEP

Un procesador de eventos complejos o CEP (De sus siglas en inglés: Complex Event Processing), es un procesador de eventos que combina los datos de varias fuentes para inferir eventos o patrones que sugieran las circunstancias más complejas.

redBorder CEP permite la ejecución de un conjunto de reglas, las cuales pueden inferir eventos, patrones y secuencias. Dispone de una API Rest a través de la cual un usuario puede añadir, eliminar o listar las reglas en tiempo real.

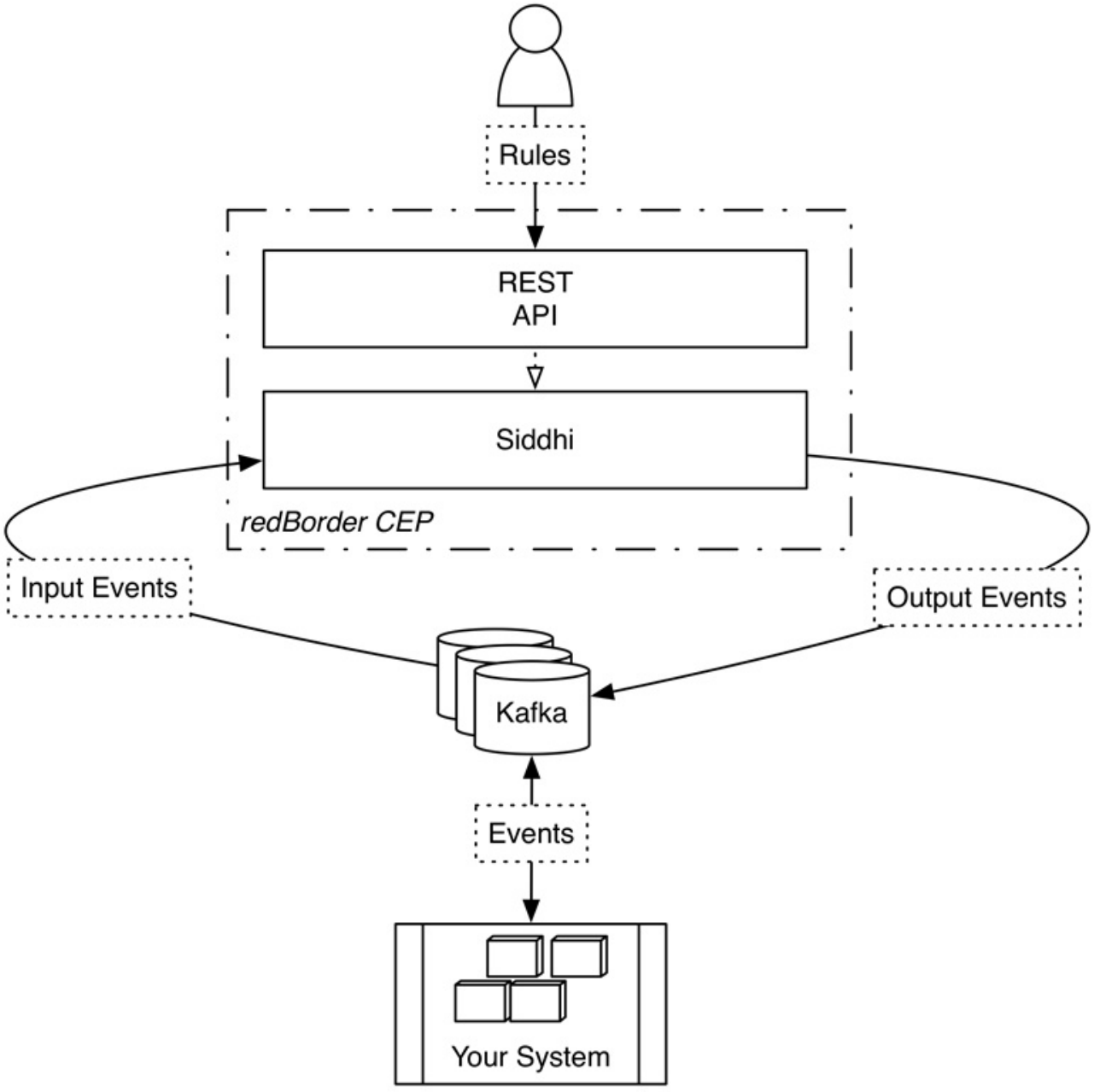

Figure 1. Arquitectura de redBorder CEP

El motor encargado de realizar las funciones del CEP es Siddhi. Siddhi CEP es un motor de procesamiento de eventos complejos en tiempo real creado por WSO2 ligero y fácil de usar escrito en Java bajo licencia Apache v2.0. Siddhi puede recibir eventos de fuentes externas y analizarlos en base a las especificaciones del usuario para posteriormente notificar los eventos apropiados.

En la arquitectura de redBorder CEP los eventos consumidos por Siddhi provienen de uno o varios topics de entrada de Kafka. El resultado de los cálculos y eventos son insertados en un nuevo topic o conjunto de topics de Kafka.

Siddhi tiene muchas características entre ellas podemos destacar:

- Funciones de agregación como sumas, máximos, mínimos, media, desviación estándar y conteo.
- Filtro y proyección de consultas usando expresiones matemáticas y lógicas.
- Valores por defecto en los atributos.
- Funciones integradas utiles para el procesamiento de eventos como, por ejemplo, ventanas de tiempo.
- Renombrado de atributos

Capítulo 2: Correlación de Eventos Complejos

En este capítulo introduciremos algunos conceptos básicos sobre la correlación de eventos complejos. En primer lugar definiremos los conceptos básicos y elementos.

Flujo de eventos

Siddhi utiliza el concepto de flujo de eventos para referirse a una secuencia de eventos ordenados en el tiempo con un esquema de atributos definido. Uno o varios de estos eventos pueden ser importados y manipulados usando consultas para identificar condiciones de eventos complejos. Estas consultas generarán nuevos eventos que son notificados como respuesta.

Un flujo de eventos está identificado por un nombre y tiene un conjunto de atributos asociados inequívocamente identificables, definiendo su esquema. Los atributos tienen un nombre único dentro del flujo de eventos y pueden ser de los siguientes tipos: `string`, `int`, `long`, `float`, `double`, `bool` u `object`.

Los eventos están asociados únicamente a un flujo y tienen un conjunto de atributos idénticos con tipos específicos. Un evento debe contener una marca de tiempo y los atributos de acuerdo a su esquema.

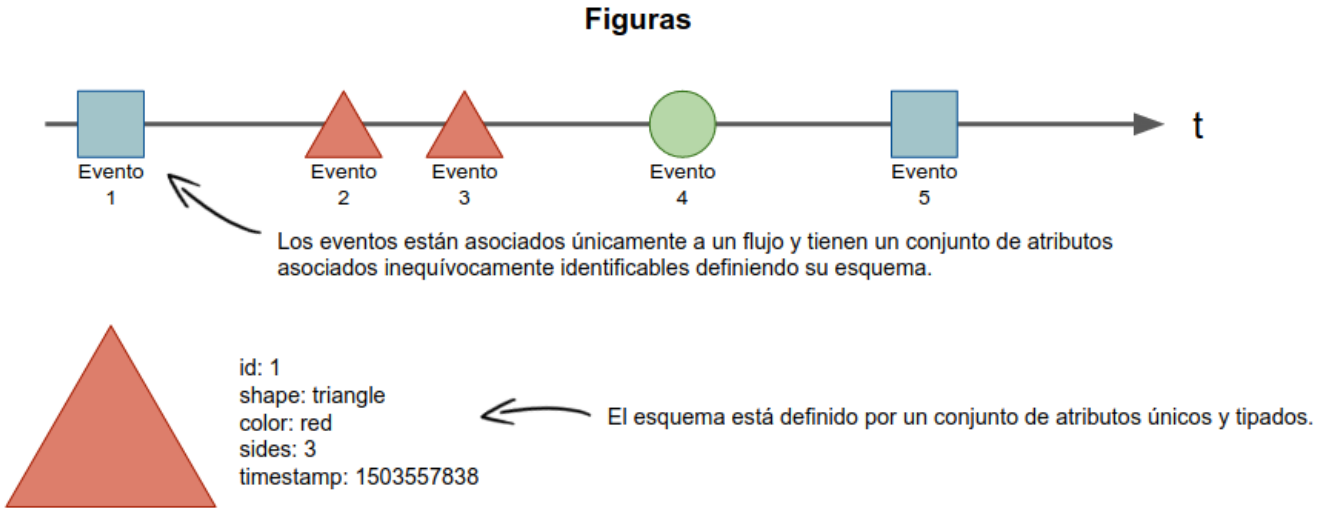

Figure 2. Flujo de eventos

Supongamos que definimos un flujo de eventos llamado "**figuras**" (Véase **Figure 1**) en el cuál se irá recibiendo una figura aleatoria a lo largo del tiempo. Un esquema válido para este flujo sería: figura = { id: int, shape: string, color: string, sides: int, timestamp: long }

Consultas

Siddhi dispone de un lenguaje DSL que permite operar con los eventos recibidos en un flujo. Con este lenguaje se pueden diseñar consultas.

Las consultas son construcciones lógicas que deriva nuevos streams al combinar streams ya existentes. Una consulta continene uno o varios flujos de eventos, manipuladores para modificar aquellos flujos de entrada, y un flujo de salida en el que publicar los eventos generados.

```
from <input stream name> select <attribute name>, <attribute name>, ... insert into <output stream name>
```

Cada consulta de Siddhi puede consumir uno o varios flujos de eventos y crea un nuevo flujo de salida a partir de ellos.

Proyecciones en las consultas

Una consulta de Siddhi tiene tres partes claramente diferenciables en sus consulta, una entrada (**from**), una salida (**insert into**) y una sección de proyección (**select**). Siddhi soporta las siguientes proyecciones:

- **Selección de los atributos necesario**

Permite seleccionar aquellos atributos del esquema definido que necesitamos.

```
from inputStream select attributeA, attributeB insert into outputStream
```

En la consulta anterior estamos seleccionando del flujo de eventos "inputStream" los atributos "attributeA" y "attributeB" y los estamos insertando en el flujo de salida "outputStream".

- **Selección de todos los atributos**

Permite seleccionar todos los atributos del esquema definido.

```
from inputStream select * insert into outputStream
```

En la consulta anterior estamos seleccionando del flujo de eventos "inputStream" todos los atributos y los estamos insertando en el flujo de salida "outputStrem". En este caso práctico la entrada es igual a la salida.

- **Renombrado de atributos**

Permite renombra los atributos seleccionados del esquema definido.

```
from inputStream select attributeA as fieldA, attributeB as fieldB insert into outputStream
```

En la consulta anterior estamos seleccionando del flujo "inputStream" los atributos "attributeA" y "attributeB" y los estamos insertando en el flujo de salida "outputStream" como atributos "fieldA" y "fieldB".

- **Introducir los valores por defecto**

Permite introducir valores por defecto, este valor sobreescribe el del atributo si este existiera.

```
from inputStream select attributeA, 'valueB' as attributeB insert into outputStream
```

En la consulta anterior estamos seleccionando del flujo "inputStream" el atributo "attributeA" y estamos insertando el atributo "attributeB" con valor por defecto "valueB" en el flujo de salida "outputStream".

- **Uso de expresiones matemáticas y lógicas**

Siddhi permite el uso de expresiones matemáticas y lógicas en sus proyecciones. En la siguiente tabla se hace referencia a ellas:

Operador	Descripción	Ejemplo
()	Alcance	`(cost + tax) * 0.5`
IS NULL	Comprueba si un atributo es nulo	is null
NOT	Negación lógica	not (price > 10)
* / %	Multiplicación, división y módulo	temp * 9/5 + 32
+ -	Suma y resta	temp * 9/5 + 32
< ⇐ > ⇐	Comparaciones: menos que, mayor o igual que, mayor que, menor o igual que	totalCost >= price * quantity
== !=	Comparaciones: Igual, distinto	totalCost >= price * quantity
AND	Conjunción lógica (Y)	temp < 40 and (humidity < 40 or humidity >= 60)
OR	Disyunción lógica (O)	temp < 40 and (humidity < 40 or humidity >= 60)

```
from TempStream select roomNo, temp * 9/5 + 32 as temp, 'F' as scale, roomNo >= 100 and roomNo < 110 as isServerRoom insert into RoomTempStream;
```

Filtros

Siddhi tiene la capacidad de aplicar filtros a sus flujos de entrada permitiendo filtrar los eventos que se van recibiendo por el criterio que deseemos.

```
from <input stream name>[<filter condition>] select <attribute name>, <attribute name>, ...
insert into <output stream name>
```

Las operaciones que pueden utilizarse para la creación de un filtro son:

- Operaciones de comparación menor que (<), menor o igual que (⇐), mayor que (>), mayor o igual que (>=), igual a (==) y distinto (**!**=).
- Operaciones lógicas de conjunción (**and**), disyunción (**or**) y negación (**not**)
- Operación de comprobación de contenido de cadena (**contains**)
- Operación de tipo de dato (**instanceof**)

from TempStream [(roomNo >= 100 and roomNo < 110) and temp > 40] **select** roomNo, temp **insert into** HighTempStream;

En el ejemplo anterior se está aplicando un filtro en el flujo de entrada **TempStream** en el cual se escifica que el número de habitación sea mayor o igual a 100 y menor a 110 y cuya temperatura sea de mayor de 40.

Ventanas

Siddhi dispone de funciones de ventanas que permiten recolectar y trabajar sobre un subconjunto limitado de eventos.

Los flujos de eventos de entradas sólo pueden tener una función de ventana asociada.

Las funciones de ventanas pueden emitir dos tipos eventos por cada evento que consumen: Los eventos actuales y los eventos que han expirado. La función de ventana emite los eventos actuales (**current-event**) cuando llega un nuevo evento y emite los eventos expirados (**expired-event**) cuando un evento expira debido al criterio de la ventana.

La salida de las funciones de ventana puede ser manipulada en base a la clasificación de los eventos.

from <input stream name>[<filter condition>]#window.<windowFunction> **select** <attribute name>, <attribute name>, ... **insert** [***expired events*** | ***current events*** | ***all events***] **into** <output stream name>

Según la clasificación de los eventos:

- **expired events**: La ventana emitirá los eventos que hayan expirado.
- **current events**: La ventana emitirá los eventos cada vez que se reciba uno.
- **all events**: La ventana emitirá tanto los eventos actuales como los que han expirado.

Agregaciones

Siddhi dispone de una serie de funciones de agregación para llevar a cabo cálculos de los valores de

los atributos indicados dentro de la ventana definida. Para ello los atributos tienen que ser de tipo `int`, `long`, `double` o `float`. Estas agregaciones son:

- sum: Calcula la suma ⟶ sum(<my-attribute>) as mySum
- average: Calcula la media ⟶ avg(<my-attribute>) as myAvg
- max: Obtiene el máximo valor ⟶ max(<my-attribute>) as myMax
- min: Obtiene el mínimo valor ⟶ min(<my-attribute>) as myMin
- count: Cuenta el número de eventos recibidos ⟶ count() as events
- stddev: Calcula la desviación estandar ⟶ stddev(<my-attribute>) as myStddev

Nótese que a las funciones de agregación se le aplica un renombrado, esto se debe a que siddhi no reconoce las funciones de agregación como atributos del flujo de eventos con el que se está trabajando.

Si no se aplicaran funciones de agregación entonces obtendríamos una salida igual a la entrada del flujo, variando únicamente en el tiempo de emisión de los eventos que depende del tipo de ventana utilzada.

Tipos de ventanas

Siddhi ofrece las siguientes funciones de ventana:

- **Ventana de tiempo**: <event> time(<int|long|time> windowTime)

Ventana deslizante que mantiene los eventos que se han recibido durante el último periodo de tiempo "**windowTime**".

Ejemplo: Calcular la suma del valor del atributo "**attrA**" de los eventos que estén dentro de la ventana de 15 segundos.

```
from inputStream#window.time(15 sec) select sum(attrA) insert into outputStream
```

- **Ventana de tiempo por lotes**: <event> timeBatch(<int|long|time> windowTime)

Ventana que procesa los eventos por lotes. Recolecta los eventos recibidos dentro del último periodo "**windowTime**" y los agrupa en un único lote.

Ejemplo: Calcular la suma del valor del atributo "**attrA**" de los eventos recolectados cada 15 segundos.

```
from inputStream#window.timeBatch(15 sec) select sum(attrA) insert into outputStream
```

- **Ventana de longitud**: <event> length(<int> windowLength)

Ventana deslizante que mantiene los últimos elementos recibidos en una ventana de tamaño "**windowLength**".

Ejemplo: Calcular la suma del valor del atributo "**attrA**" de los 5 últimos evetnos recibidos.

```
from inputStream#window.length(5) select sum(attrA) insert into outputStream
```

- **Ventana de longitud por lotes**: <event> lengthBatch(<int> windowLength)

Ventana deslizante que emite los eventos como un lote a la llegada del i-ésimo evento en una ventana de tamaño "**windowLength**".

Ejemplo: Calcular la suma del valor del atributo "**attrA**" cada 5 eventos recolectados.

```
from inputStream#window.lengthBatch(5) select sum(attrA) insert into outputStream
```

- **Ventana deslizante de tiempo externo**: <event> time(<long> timestmap, <int|long|time> windowTime)

Ventana deslizante que trabaja con el tiempo proporcionado por el flujo de eventos de entrada en lugar de utilizar el de la máquina.

Ejemplo: Calcular la suma del valor del atributo "attrA" utilizando una ventana de tiempo de 15 segundos y cuya marca de tiempo vendrá determinada por el campo "timestamp" del flujo de eventos entrante que está procesando.

```
from inputStream#window.externalTime(timestamp, 15 sec) select sum(attrA) insert into outputStream
```

- **Ventana deslizante con cron**: <event> cron(<string> cronExpression)

Ventana deslizante que procesa los eventos en base a un patrón de tiempo basado en una expresión de cron.

Ejemplo: Calcular la suma del valor del atributo "attrA" cada 5 segundos.

```
from inputStream#window.cron('*/5 * * * * ?') select sum(attrA) insert into outputStream
```

- **Ventana deslizante de primeras únicas ocurrencias**: <event> firstUnique(<string> attribute)

Ventana que mantiene únicamente los primeros eventos que son únicos en base a un atributo especificado.

Ejemplo: Obtener el primer evento de todas las IPs.

```
from inputStream#window.firstUnique(ip) select * insert into outputStream
```

- **Ventana deslizante de últimas ocurrencias únicas**: <event> unique(<string> attribute)

Ventana que mantiene los últimos eventos que son únicos en base a un atributo especificado.

Ejemplo: Obtener el último evento único de todas las IPs.

```
from inputStream#window.unique(ip) select * insert into outputStream
```

- **Ventana deslizante de ordenamiento**: <event> sort(<int> windowLength, <string> attribute, <string> order, .. , <string> attributeN, <string> orderN)

Ventana que ordena los atributos especificados de forma ascendete o descendente.

Ejemplo: Mantener 5 eventos ordenados por precio de forma ascendente.

```
from inputStream#window.sort(5, price, asc) select * insert into outputStream
```

- **Ventana deslizante de frecuencia**: <event> frequent(<int> eventCount, <string> attribute, .. , <string> attributeN)

Ventana que devuelve los últimos eventos más frecuentes recibidos. El cálculo de la frecuencia se basa en el algoritmo de contenio Misra-Gries.

Ejemplo: Obtener los 3 eventos más frecuentes.

```
from inputStream#window.frequent(3) select * insert into outputStream
```

- **Ventana deslizante de frecuencia con pérdidas**: <event> lossyFrequent(<double> supportThreshold, <double> errorBound)

Ventana que identificará y devolverá todos los eventos cuya frecuencia exceda el valor "**supportThreshold**". El cálculo de la frecuencia se basa en el algoritmo de *Lossy Counting*

```
from inputStream#window.lossyFrequent(0.1, 0.01) select * insert into outputStream
```

Capítulo 3: Patrones y secuencias

En este capítulo hablaremos sobre los patrones y secuencias, cuáles son las principales diferencias y cómo podemos crear reglas acorde a ellos.

Patrones

Los patrones permiten a los flujos de eventos correlar en el tiempo y detectar patrones basados en un orden de llegada. Con los patrones puede haber otros eventos entre aquellos que hacen *match* con nuestra regla. Los patrones pueden correlar los eventos de múltiples fuentes o de una única fuente, por lo tanto cada evento que haga *match* necesita ser referenciado para que pueda ser accedido en el futuro para su procesado y generación de su salida.

La sintaxis de un patrón es como sigue a continuación:

> **from** {**every**} <input event reference>=<input stream name>[<filter condition>] -> {**every**} <input event reference>=<input stream name>[<filter condition>] -> ... **within** <time gap> **select** <input event reference>.<attribute name>, <input event reference>.<attribute name>, ... **insert into** <output stream name>

En esta sintaxis podemos apreciar varios detalles:

- En la expresión `{every} <input event reference>=<input stream name>[<filter condition>] ->`:
  - La palabra clave `every` indica si queremos que el patrón sólo haga *match* una única vez (omitiendo dicha palabra clave). Hay que usar apropiadamente esta palabra reservada, ya que si la utilizamos, el patrón hará *match* por cada evento que cumpla la condición.
  - `<input event reference>=` es la referencia del evento que ha hecho *match* para su posible uso en el futuro.
  - `<input stream name>[<filter condition>]` es el flujo de eventos de entradas. Pueden aplicarse filtros para excluir aquellos eventos que no cumplan con las condiciones que se especifiquen.
  - `->` con este operador podemos correlar la llegada de los eventos entrantes, pudiendo tener varios eventos entre aquellos que hacen *match*.

Hay que tener en cuenta que a los flujos de entradas no se le pueden asociar ventanas de tiempo.

Para establecer un intervalo de tiempo en el que evaluar el patrón se utiliza la expresión `within <time gap>`. `<time gap>` es una expresión que utiliza el tipo especial de datos `Time`. `Time` es un tipo compuesto por un entero (`int`) y una unidad (`unit`) de tiempo, el tipo de datos `Time` realmente devuelve un número de tipo `long`, así que simplemente es un tipo que ayuda al usuario a definir intervalos de tiempo de una forma más cómoda y natural. En la siguiente tabla podemos observar los tipos de unidades así como ejemplos:

Unidad	Sintaxis	Ejemplo	Devuelve
Año	year | years	1 year	217728000000
Mes	month | months	1 month	18144000000
Semana	week | weeks	1 week	604800000
Day	day | days	1 day	86400000
Hour	hour | hours	1 hour	3600000
Minutos	minute | minutes | min	1 min	60000
Segundos	second | seconds | sec	1 sec	1000
Milisegundos	millisecond | milliseconds	1 millisecond	1

El tipo `Time` no está limitado a sólo el uso de un tipo de unidad, sino que puede combinarse con varias unidades para hacer intervalos más precisos, como por ejemplo: `1 hour 30 min` para indicar una hora y media devolviendo como resultado `5400000`.

Hay que tener en cuenta que cuando se utiliza la expresión `within <time gap>` estamos estableciendo un intervalo en el cual todos los patrones tienen que hacer *match*. En caso contrario todos los eventos que han hecho *match* y a los cuales se tienen referencia serán descartados, comenzando de nuevo con el proceso.

Para asentar mejor lo explicado, vamos a analizar el siguiente patrón de ejemplo:

> **from every** (e1=TempStream) -> e2=TempStream[e1.roomNo==roomNo **and** (e1.temp + 5) <= temp] **within** 10 min **select** e1.roomNo, e1.temp **as** initialTemp, e2.temp **as** finalTemp **insert into** AlertStream;

En primer lugar tenemos la expresión `every e1=TempStream`, con la que simplemente estamos capturando los eventos del flujo `TempStream` y los estamos referenciando con `e1`. Podemos observar que tenemos la palabra clave `every` indicando de esta forma que este patrón puede hacer *match* en más de una ocasión. Como no hay filtros aplicados este patrón hará *match* por cada evento que llegue de entrada y estos serán almacenados con sus correspondientes referencias.

En segundo lugar tenemos la expresión `e2=TempStream[e1.roomNo==roonNo and (e1.temp + 5) <= temp ]` en este caso estamos capturando un evento que cumpla con el siguiente filtro:

- Si el número de habitación del último evento recibido del primer patrón (`e1`) coincide con el número de habitación del segundo evento (Para hacer uso de los atributos de los eventos capturados no se utiliza ningún tipo de referencia)
- Y si la temperatura ha incrementado en 5 grados con respecto al último evento recibido del primer patrón pero es menor o igual que el del propio evento capturado.

Entonces el patrón completo hace *match*, obteniendo el número de habitación, la temperatura inicial y la temperatura final inyectando los datos en el flujo de salida `AlertStream`. Sin embargo debemos de tener en cuenta la expresión `within 10 min` que nos indica que todas estas condiciones tienen que cumplirse en un intervalo de tiempo de 10 minutos.

Por lo tanto esta regla se leería de la siguiente forma: ***Alerta cuando la temperatura de una habitación incremente 5 grados en un intervalo de 10 minutos***

A continuación veremos dos tipos de patrones que pueden utilizarse para dar más flexibilidad a este tipo de correlación.

Patrones lógicos

Los patrones no tienen por qué hacer *match* cuando llegan los eventos en un orden temporal, en ocasiones necesitaremos relaciones lógicas para definir el comportamiento de un patrón y así ofrecer más flexibilidad.

Las palabra claves como `and` (conjunción lógica "Y") y `or` (disyunción lógica "O") pueden ser utilizadas en lugar de "->" para ilustrar relaciones lógicas.

- Con el operador `and` deben de cumplirse las condiciones de dos eventos para que el patrón pueda hacer *match*.
- Con el operador `or` debe de cumplirse al menos una de las condiciones para que el patrón pueda hacer *match*.

Veamos el siguiente ejemplo:

> **from every** (e1=RegulatorStream) -> e2=TempStream[e1.roomNo==roomNo **and** e1.tempSet <= temp] **or** e3=RegulatorStream[e1.roomNo==roomNo] **select** e1.roomNo, e2.temp **as** roomTemp **having** e3 **is** null **insert into** AlertStream;

Este ejemplo se parece mucho al visto en la sección anterior, sin embargo podemos ver la siguiente expresión: `e2=TempStream[e1.roomNo==roomNo and e1.tempSet <= temp ] or e3=RegulatorStream[e1.roomNo==roomNo]`

En dicha expresión estamos evaluando que si la habitación es la mismas para el evento del primer patrón y la temperatura es menor o igual que el evento del segundo patrón o si por el contrario no lo es pero en cambio sí coincide que la habitación es la misma para el evento del tercer patrón. Entonces hace *match* y se dispara.

Sin embargo debemos de tener en cuenta la expresión `having e3 is null`. La palabra reservada `having` nos permite filtrar los eventos después de una agregación o de un procesado de eventos. Así finalmente estamos añadiendo un filtro en el cual estamos especificando que el evento del tercer patrón (referenciado como `e3`) tiene que ser nulo, condición que se cumple si el primer patrón y el segundo patrón se cumplen. En caso contrario no se disparará el patrón.

Patrones de conteo

Los patrones de conteo nos permiten hacer *match* de múltiples eventos que contemplan la misma condición. El número de eventos esperados puede ser limitado usando los siguientes operadores:

- Con `<N:M>` contará de `N` a `M` eventos, así por ejemplo `<1:4>` contará de 1 a 4 eventos.
- Con `<N:>` contará a partir de `N` eventos, así por ejemplo `<2:>` contará a partir de 2 eventos.

- Con `<:M>` contará hasta `M` eventos, así por ejemplo `<:5>` contará hasta 5 eventos.
- Con `<X>` contará exactamente `X` eventos, así por ejemplo `<5>` contará exactamente 5 eventos.

La diferencia entre `<:M>` y `<X>` es que con el operador `<X>` estamos indicando que el patrón se tiene que cumplir exactamente X veces para que ocurra un *match* mientras que con el operador `<:M>` estamos indicando que se puede cumplir hasta M veces pero haciendo *match* por cada vez.

Observemos la siguiente regla:

```
from every (e1=RegulatorStream) -> e2=TempStream[e1.roomNo==roomNo]<1:> -> e3=RegulatorStream[e1.roomNo==roomNo] select e1.roomNo, e2[0].temp - e2[last].temp as tempDiff insert into TempDiffStream;
```

En ella tenemos la siguiente expresión `e2=TempStream[e1.roomNo==roomNo]<1:>` de conteo definido, para este caso estamos poniendo como limitante que haya uno o más eventos para poder hacer *match*.

Secuencias

Las secuencias, al igual que los patrones, permiten correlar sobre los eventos recibidos y detectar secuencias de eventos según su orden de llegada. A diferencia de los patrones, en una secuencia no puede haber otros eventos entre aquellos que hacen *match* con nuestra regla. Las secuencias pueden currerlar los eventos de múltiples fuentes o de una única fuente, por lo tanto cada evento que haga *match* necesita ser referenciado para que pueda ser accedido en el futuro para su procesado y generación de su salida.

La sintaxis de una secuencia es como sigue a continuación:

```
from {every} <input event reference>=<input stream name>[<filter condition>], <input event reference>=<input stream name>[<filter condition>]{+|*|?}, ... within <time gap> select <input event reference>.<attribute name>, <input event reference>.<attribute name>, ... insert into <output stream name>
```

Al igual que ocurre con los patrones, las secuencias no pueden tener una ventana asociada. Podemos observar que hay una gran similitud en cuanto a los patrones con la única diferencia en que el delimitador es una coma (`,`). El resto de elementos son totalmente análogos al uso de patrones salvo el conteo que se explicará en sucesivas secciones.

Secuencias lógicas

Las secuencias no sólo hacen *match* en eventos consecutivos sino que pueden ser correlados con relaciones lógicas. Para ello podemos utilizar las palabras claves `and` y `or` del mismo modo que se utilizand en los patrones.

Por ejemplo

```
from every e1=RegulatorStream, e2=TempStream and e3=HumidStream select e2.temp, e3.humid insert into StateNotificationStream;
```

En el ejemplo anterior estamos indicando que cuando llegue un evento del flujo de eventos del regulador y seguidamente llegue un evento del sensor de temperatura y de humedad, obtengamos la temperatura y humedad de cada evento y lo notifiquemos.

Tengamos en cuenta nuevamente que es necesario que los eventos sean consecutivos no pudiendo existir otros eventos que no sean relevantes entre cada *match* de la regla.

Secuencias de conteo

Las secuencias nos permite hacer *match* con múltiples eventos consecutivos en base a unas condiciones. A diferencia de los patrones, las secuencias tiene otros limitadores de eventos que se resumen a continuación:

- Con * estamos indicando ninguno o muchos.
- Con + estamos indicando uno o muchos.
- Con ? estamos indicando ninguno o uno.

Tal y como ocurre con los patrones, se puede hacer referencia a los eventos capturados utilizado los índices o la palabra clave `last`.

```
from every e1=TempStream, e2=TempStream[e1.temp <= temp]+, e3=TempStream[e2[last].temp > temp] select e1.temp as initialTemp, e2[last].temp as peakTemp insert into TempDiffStream;
```

En el ejemplo anterior podemos ver la expresión `e2=TempStream[e1.temp <= temp]+` donde estamos indicando que se reciban uno o más eventos.

Capítulo 4: Particiones

Con las particiones Siddhi puede dividir tanto los eventos de entradas como las consultas y procesarlos en paralelo. Cada partición será etiquetada por una clave y todos aquellos eventos correspondientes a dicha partición serán procesados. Las particiones pueden tener más de una consulta.

Las claves de particionado se pueden clasificar en dos tipos:

- Particionado por variable: Las cuales utilizan un atributo de cadena (`string`)
- Particionado por rango: Las cuales utilizan atributos numéricos

Particionado por variable

Como se ha indicado anteriormente este tipo de particionado utiliza atributos de cadena para particionar. Su sintaxis es la siguiente:

> **partition with** (<attribute name> **of** <stream name>, <attribute name> **of** <stream name>, ...) **begin** <query> <query> ... **end**;

Con la expresión `<attribute name> of <stream name>` estamos definiendo qué atributo de qué flujo de eventos se utilizará para el particionado.

Dentro de una partición puede haber varias consultas que son iguales para cada partición. Veamos un ejemplo:

> **partition with** (deviceID **of** TempStream) **begin from** TempStream#window.length(10) **select** roomNo, deviceID, max(temp) **as** maxTemp **insert into** DeviceTempStream **end**;

En este ejemplo estamos haciendo un particionado por ID de dispositivo, si tenemos tres dispositivos tendremos un total de tres particiones que en paralelo realizarán la consulta para cada dispositivo de forma independiente.

Particionado por rango

El particionado por rango utiliza atributos numéricos para hacer el particionado. Su sintaxis es la siguiente:

> **partition with** (<condition> **as** <partition key> **or** <condition> **as** <partition key> **or** ... **of** <stream name>, ...) **begin** <query> <query> ... **end**;

En este tipo de particionado podemos observar la siguiente expresión `<condition> as <partition key> or <condition> as <partition key> or ⋯ of <stream name>` donde:

- `<condition> as <partition key>`: Se trata de una condición que debe de cumplirse, pueden utilizarse operadores lógicos `or` y `and`. si se observa la condición es renombrada para su uso como clave de particionado.
- expresiones `or`: Sirven para concatenar las claves de particionado.
- `of <stream name>`: Para designar al flujo de eventos al cual pertenecen las claves de particionado.

```
partition with ( roomNo>=1030 as 'serverRoom' or roomNo<1030 and roomNo>=330 as 'officeRoom' or roomNo<330 as 'lobby' of TempStream) ) begin from TempStream#window.time(10 min) select roomNo, deviceID, avg(temp) as avgTemp insert into AreaTempStream end;
```

En este ejemplo estamos haciendo un particionado por habitaciones, veamos qué estamos definiendo exactamente:

- `roomNo>=1030 as 'serverRoom'`: Estamos particionando por `serverRoom` que son aquellas cuyo número es mayor de `1030`.
- `roomNo<1030 and roomNo>=330 as 'officeRoom'`: Estamos particionando por `officeRoom` que son aquellas habitaciones cuyo número está comprendido entre `1031` y `330`
- `roomNo<330 as 'lobby'`: Estamos particionando por `lobby` que son aquellas habitaciones cuya numeración están por debajo de `330`.

Así que básicamente estamos particionando por `serverRoom`, `officeRoom` y `lobby` pertenecientes al flujo de eventos `TempStream`.

Con este ejemplo podemos ver la utilidad de utilizar rangos, ya que en vez de particionar por cada habitación (que tendría un coste computacional alto) estamos particionando por rangos, haciendo sólo tres tipos de particionados (uno por cada tipo de habitación).

Flujos internos

En ocasiones tendremos la necesidad de utilizar flujos que no están definidos para su uso y que pueden servir para, por ejemplo, dividir las consultas en pasos en caso de cumplir ciertas condiciones.

Siddhi tiene la capacidad de generar flujos internos. Estos flujos internos suelen ser usados dentro de las particiones para comunicar unas consultas con otras de la misma partición. Los flujos internos comienzan con una almohadilla (`#`).

Los flujos internos no pueden accedidos fuera del bloque de particionado, es decir, es necesario que los flujos internos pertenezcan a una partición sin posibilidad de intercambiar información entre particiones.

```
partition with ( deviceID of TempStream ) begin from TempStream#window.time(1 min) select roomNo, deviceID, temp, avg(temp) as avgTemp insert into #AvgTempStream; from #AvgTempStream[avgTemp > 20]#window.length(10) select roomNo, deviceID, max(temp) as maxTemp insert into deviceTempStream end;
```

En este ejemplo estamos particionando por ID de dispositivo y estamos aplicando flujos internos para utilizar dos consultas:

- En la primera consulta estamos obteniendo el número de habitación, el ID del dispositivo, la temperatura en el instante y la temperatura media y lo estamos inyectando en el flujo `#AvgTempStream`.
- En la segunda consulta estamos extrayendo los eventos de `#AvgTempStream` y filtrando aquellos cuya temperatura media es mayor a `20`, para acumularlos hasta obtener `10` para posteriormente obtener el número de habitación, ID de dispositivo y la temperatura máxima.

Como puede observarse el uso de flujos internos puede ser interesante para llevar a cabo ciertas operaciones y consultas.