CEP

USE

GUIDE

AUGUST 2014

indra

VERSION CONTROL

Make sure this document is up to date. Printed or local copies may be obsolete.

1 INDEX

2 INTRODUCTION

2.1 Goals and scope of this document

This document is intended to be a reference to use the capabilities of the CEP in the Sofia2 platform, and to know its configuration mechanisms.

3 CEP

3.1 What a CEP Provides

Event-based publishing and subscription systems have always been part of architectures related with system integration through MOM (Message Oriented Middleware) platforms. More recently, SOA architectures have been integrated within an ESB (Enterprise Service Bus) infrastructure.

As well, for a few years, BAM (Business Activity Monitoring) systems allow to process business events from different sources and notify them so the monitoring agents make a decision based on the event context.

Therefore, the current challenge is not the generation, capture or notification of such events, but the ability to process a great amount of real-time events, to correlate these events generating automatic answers based on the event semantic and, last but not least, to integrate all of it in a SOA architecture.

The CEP goal is the capture and processing of different type events in an unordered manner, in the so-called “cloud of events”. We say that a cloud of events may contain several streams, being a stream a special case of cloud, where all the events are of the same type.

3.2 Adopted Technology

From all the possible CEP solutions existing in the market, we have chosen Siddhi, a product that is part of the WSo2 platform. Siddhi is the core upon which the CEP solution for Sofia2 has been built.

The decision has been motivated by the following features:

- The product is part of WSo2 and will be maintained by it.
- It doesn’t make use of data structures to process the information, which makes it an optimal (non-redundant) solution to handle the JSON-based Sofia2 data structures.
- It offers a high performance, processing more than 2.5M events/second in a commodity hardware server.
- It supports scalability through Hazelcast, the GRID solution adopted by Sofia2.

4 CEP ENGINE INTEGRATED IN THE SIB

4.1 Data Model

4.1.1 Event Definition

The definition of an event is the relationship between an ontology and its decomposition in elements meaningful to the CEP. When we define an event, we are extracting a portion of an ontology and mapping it to an Attribute → Type list.

```
{
    "Wattmeter" : {
        "identifier" : "W-TA3231-1" ,
        "timestamp" : 1358086507725 ,
        "measure" : 30 ,
        "unit":"W",
        "accumulatedEnergy" : 238,
        "accumulatedEnergyUnit" : "Wh",
        "active" : true,
        "gpsCoordinate" : {
            "height" : true ,
            "latitude" : 40 ,
            "longitude" : -3.67495
          }
      }
}
```

If we take as reference the Ontology instance represented in the above data, we may create an Event Definition like the following one:

```
Wattmeter__identifier String
Wattmeter__gpsCoordinate__latitude Int
```

So we have an Event Definition that will work with two values of the Ontology, the identifier and the latitude. We can create as many definitions as we need on the same Ontology.

Data types supported by CEP are:

string
float
int
boolean

The format for defining the data is:

[Attribute Name] [Space] [Data Type] [Comma] [Attribute Name] [Space] [Data Type] …

ID	DEFINICION	ONTOLOGIA_ID	IDENTIFICACION
0e218b1c-82...	Watorimetro__medida float	4	Wato
NULL	NULL	NULL	NULL

The above record defines an Event identified as **Wato** that is related with Ontology with ID 4 (Wattmeter), based on a single attribute of type float named Wattmeter__measure.

```
{
    "Wattmeter" : {
        "identifier" : "W-TA3231-1" ,
        "timestamp" : 1358086507725 ,
        "measure" : 30 ,
       …
```

4.1.2 CEP Rule

Once we have created Event Definitions, we can create CEP Rules using them. This will make the CEP launch queries over the received data.

ID	CONDICION	PROYECCION	DESTINO
948...	WATO [WATORIMETRO__MEDIDA>30]	WATORIMETRO__MEDIDA AS MEDIDA	WATORIMETROMEDIDA
c16...	every E1=EVENTOSHCEP [SENSORHUMEDA...	E1.SENSORHUMEDADCEP__MEDIDA AS...	TEMPHUMPATTERN
NULL	NULL	NULL	NULL

In this table we define the structure of a Query that makes use of the Events defined in the previous step.

CONDICION
WATO [WATORIMETRO__MEDIDA>30]

The CONDICION column defines the validations over the input data (defined Events, or results from other queries). In this case, the input is the **Wato** Event, and the data is filtered for the condition **Wattmeter_measure > 30**, that is, for measures greater than 30.

PROYECCION
WATORIMETRO__MEDIDA AS MEDIDA

The PROYECCION column defines the output data, based on the input data. In this case, the attribute **Wattmeter_measure** from the input will be called simply **Measure** in the output.

DESTINO
WATORIMETROMEDIDA

The DESTINO column identifies the new Event generated with the filtered output data. In this case, it will be called **Wattmetermeasure**.

So, for this example, every time an instance of the Wattmeter Ontology is received, the value of the measure field is obtained and passed to the CEP as a **Wato** Event.

The CEP evaluates this value internally using the defined Queries. If any of these instances has a measure greater than 30, the condition is fulfilled.

This generates a new **Wattmetermeasure** Event, that we could use in turn as an input for more complex queries. In this Event, the value of the **Wattmeter_measure** attribute is stored with the name **Measure**.

> **Notice**
>
> The CEP engine, Siddhi, is case sensitive, so the reserved keywords (data types, query tokens) must always be lowercase.
>
> On the contrary, to avoid mistakes in the definition of the Queries and enhance their legibility, we have chosen all the identifiers (either from the Event Definitions, or created in the Queries like Stream, Attribute or Join identifiers) to be uppercase.

4.2 Runtime

4.2.1 Event Execution

When an Ontology instance is inserted in the SIB, the processor delegates in the CEP service.

```
public void addEventCep(String ontology, String json)
```

Event and Rule definitions are initialized when the CEP module is started, so the events and rules are available and we can make use of the relationships between them.

The engine receives the Ontology instance and gets the necessary values according to the Event Definitions, creating a new Event instance for every configured Event Definition.

Continuing with our example, when the CEP receives an instance of the **Wattmeter** ontology, it gets the **measure** value, of type **int**, as specified in the **Wato** Event Definition.

Every data insertion is analyzed asynchronously by the CEP, firing events when the conditions defined in the Query are fulfilled, in this case, when **measure** is greater than 30.

4.3 Subscription to Events

KPs can subscribe to Events fired by the CEP. They only have to indicate that the subscription is of type CEP and the name of the Rule (generated event) they want to subscribe to.

Every time the Rule is met and its event is generated, the associated data will be notified to the subscribed KPs through an Ontology with the following structure:

```
{
```

```
"Subscription" : "Rule name",
"inEvents" : [ {"key":"value", "key": "value", …},{"key":"value", "key": "value", …},…]
"removeEvents" : [ {"key":"value", "key": "value", …},{"key":"value", "key": "value", …},…]
}
```

Continuing with our example, KPs could subscribe to the event named **WATTMETERMEASURE**.

4.4 Event extension

In order to extend the event behavior, there are plugins of type `EventPlugin`, described in the document "SOFIA2 – Configuration, Extension and Custom SIB Guide".

These plugins have to implement the following method, defined by the `EventPlugin` Interface:

```
void receive(String event, long timeStamp, List<Map<String, Object>> inEvents, List<Map<String, Object>> removeEvents);
```

These plugins are Spring Beans whose name has to match the name of the generated event (uppercase) followed by "CEP".

In our example, the Bean name has to be **WATTMETERMEASURECEP**.

It will receive the following data: the name of the generated event; a list of the events added to the CEP, being each element a Map with the names and values of the inserted attributes; and an analogous list of the events removed from the CEP.

5 WORKFLOW (CEP)

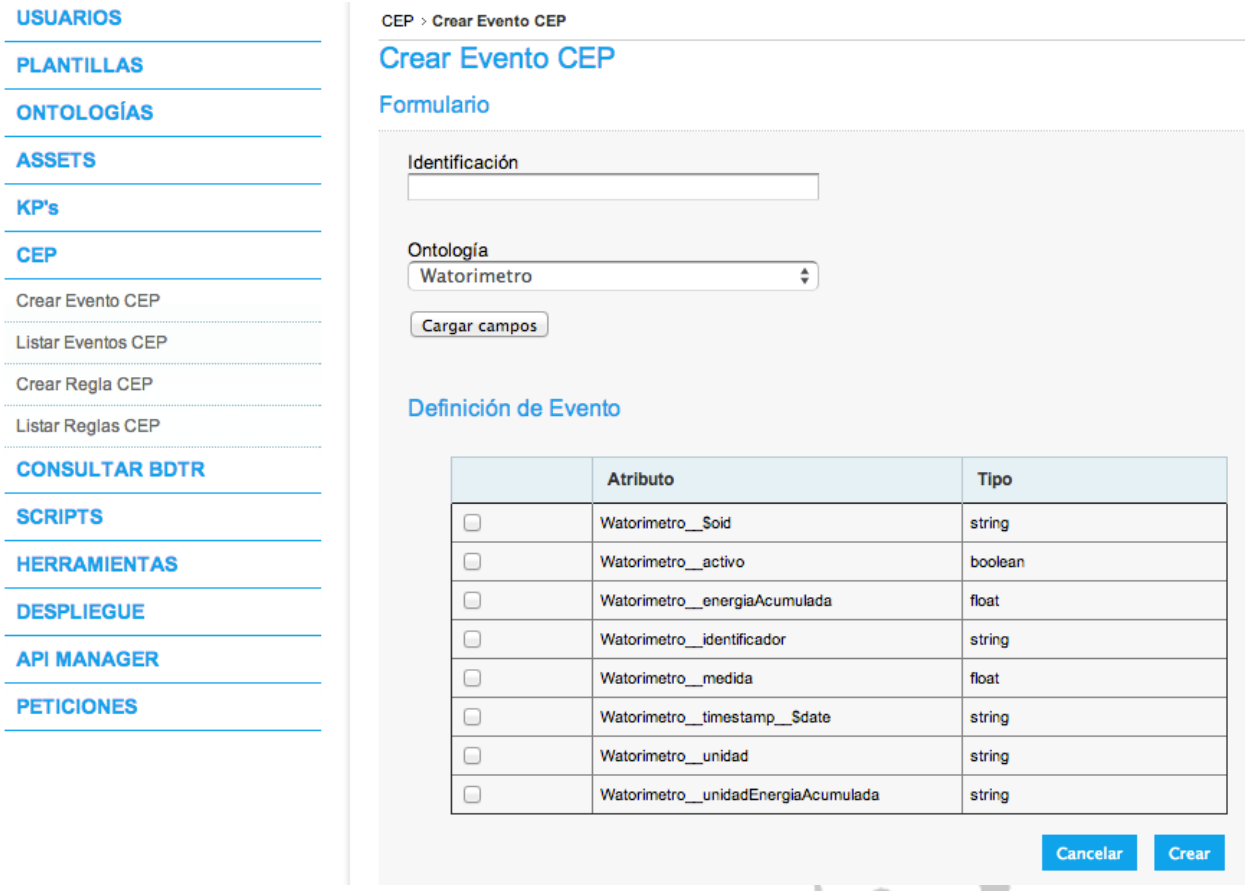
5.1 Event Definition

5.1.1 New Event Definition in the platform

An Event Definition must be registered in the platform, before it can be used to created CEP Rules, as explained above.

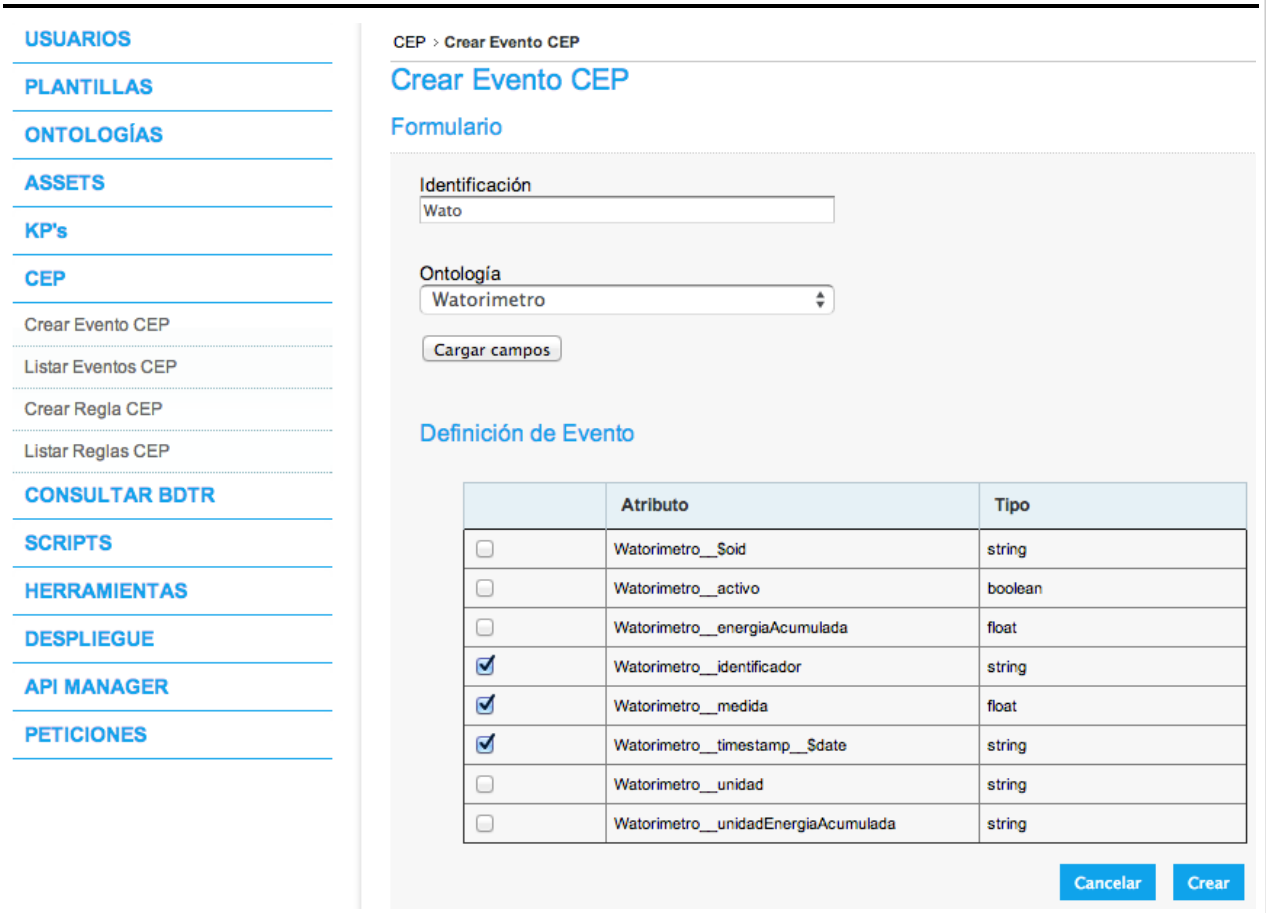
In the console, there is a **CEP** section where you can create and edit event definitions.

Let's define our "**Wato**" Event.

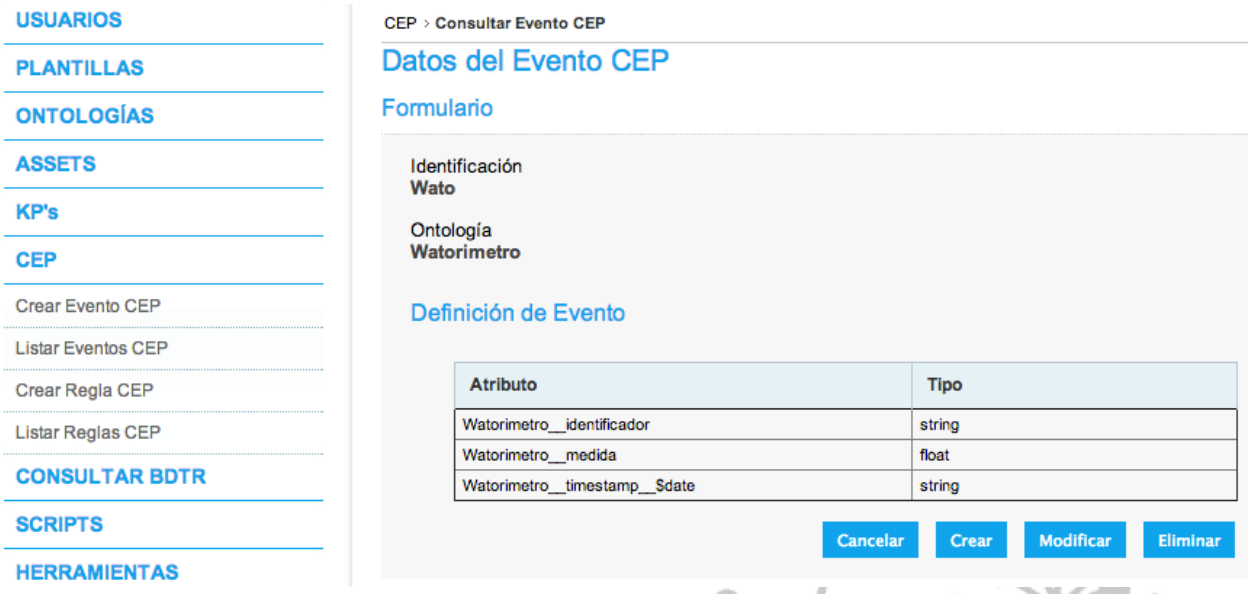


We select an Ontology for the creation of the Event Definition, and we push the "Load fields" button. We get a table with all the Ontology attributes and their types.

We must select at least one attribute for the Event Definition.



Once we have entered an identifier and selected some attributes, we push the “Create” button.



The “Wato” Event Definition has been registered and we can check it and manage it through the console.

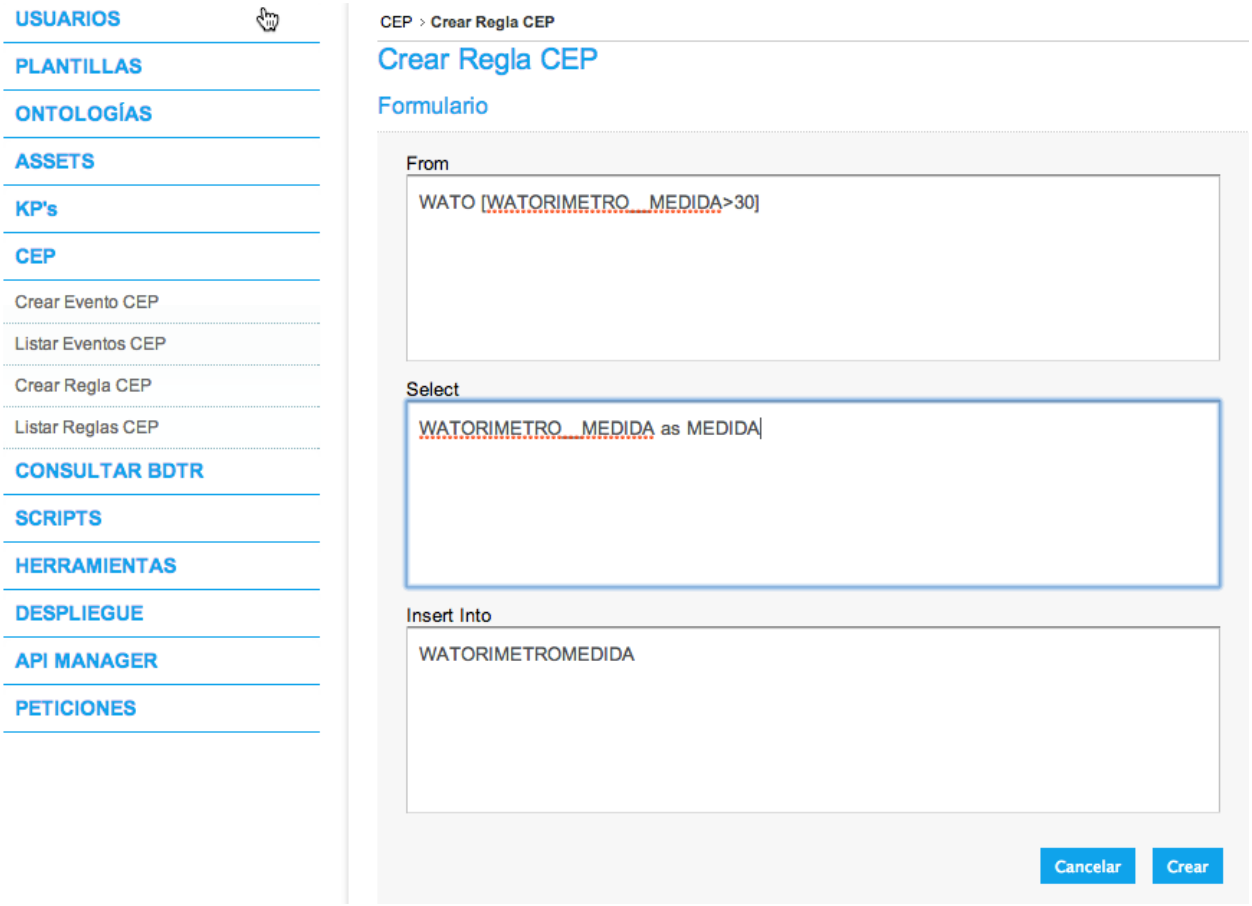
5.2 CEP Rules

5.2.1 New CEP Rule in the platform

A CEP Rule must be registered in the platform, before it can be used.

In the console, there is a **CEP** section where you can create and edit CEP Rules.

Let's create our "**WATTMETERMEASURE**" Rule.



We build the rule according to the patterns explained above, using the existing Events or Rules that we have permissions over.

We have to bear in mind that the oputput Event for a CEP Rule is unique and its name will be equal to the Rule identifier.

We push the "New" button.

The CEP Rule “WATTMETERMEASURE” has been registered and we can check it and manage it through the console.

6 CEP API

Through the cepService, you can obtain a reference to the cepManager, which in turn gives you Access to the Siddhi API. So we can make use of the CEP in our platform plugins or extensions.

This API is configured to interact with the platform and we'll always get a reference to a previously configured context.

To access this API, we'll inject the CEP service in our code:

```
@Autowired
private CepService cepService;
```

That will allow us to add data to the CEP about Event Definitions registered in the platform:

```
cepService.addEventCep(String ontologia, String json)
```

Or to Access the low-level API that will allow us to define the Streams (Event Definition), Queries (Instance Definition) and Callbacks (Event Execution).

```
cepService.getCepManager()
```

The more significant methods in the low-level API are:

- Methods to define a Stream

  ```
  public InputHandler defineStream(StreamDefinition streamDefinition)
  ```

  ```
  public InputHandler defineStream(String streamDefinition)
  ```

- Method to remove a Stream

  ```
  public void removeStream(String streamId)
  ```

- Method to check the existence of a Stream

  ```
  boolean checkEventStreamExist(StreamDefinition newStreamDefinition)
  ```

- Methods to define a Query

  ```
  public String addQuery(String query)
  ```

  ```
  public String addQuery(Query query)
  ```

- Method to remove a Query

  ```
  public void removeQuery(String queryId)
  ```

- Method to get a defined Query

  ```
  public Query getQuery(String queryReference)
  ```

- Methods to add a Cabllback

  ```
  public void addCallback(String streamId, StreamCallback streamCallback)
  ```

  ```
  public void addCallback(String streamId, StreamCallback streamCallback)
  ```

7 Siddhi Language Query.

7.1.1.1 Filters

from <stream-name> {<conditions>} insert into <stream-name> ({<attribute-name>}| '*'|)

Filter query creates an output stream and inserts any events from the input stream that satisfies the conditions defined with the filters to the output stream. Filters support following types of conditions

1. >, <, ==, >=, <=, != 2. contains, instanceof 3. and, or, not

Following example shows sample queries for filters.

Stream *defined by "**define stream StockExchangeStream (symbol string, price int, volume float);**" is used in below queries.*

>, <, == , >=, <=, !=

From all events of the StockExchangeStream stream, output only the events having price >= 20 and price < 100 to the StockQuote stream, where the output event will only have symbol and volume as its attributes.

from StockExchangeStream[price >= 20 and price < 100] insert into StockQuote symbol, volume

Here we are only projecting 'symbol' and 'volume' as the output of the query, and hence the output stream 'StockQuote' will only contain 'symbol' and 'volume' attributes, If the projection is omitted or '*' is used, the query will output all the attributes of input streams.

From all events of the StockExchangeStream stream, output only the events having volume > 100 and price != 100 to the StockQuote stream, where the output event will have symbol, volume and price as its attributes.

from StockExchangeStream[volume > 100 and price!=10] insert into StockQuote symbol, volume, price

The output stream 'StockQuote' will contain 'symbol', 'price' and 'volume' attribute,

contains, instanceof

From all events of the StockExchangeStream stream, output only the events where symbol is an instance of java.lang.String to the StockQuote stream, where the output event will only have symbol and price as its attributes.

from StockExchangeStream[symbol instanceof 'string'] insert into StockQuote symbol, price

The output stream 'StockQuote' will only contain 'symbol' and 'price' attributes, If the projection is omitted or '*' is used, the query will output all the attributes of input streams.

Other than that we can also use *instanceof condition for 'float', 'long' , 'integer', 'double' and 'boolean'.*

From all events of the StockExchangeStream stream, output only the events where symbol contains 'ws' to the StockQuote stream, where the output event will only have symbol and price as its attributes.

from StockExchangeStream[symbol contains 'ws'] insert into StockQuote symbol,

price

The output stream ‘StockQuote’ will only contain ‘symbol’ and ‘price’ attributes, If the projection is omitted or ‘*’ is used, the query will output all the attributes of input streams.

contains condition can only be applied to strings.

and, or, not

From all events of the StockExchangeStream stream, output only the events having price >= 20 and price < 100 to the StockQuote stream, where the output event will only have symbol and volume as its attributes.

from StockExchangeStream[price >= 20 and price < 100] insert into StockQuote symbol, volume

Here we are only projecting ‘symbol’ and ‘volume’ as the output of the query, and hence the output stream ‘StockQuote’ will only contain ‘symbol’ and ‘volume’ attribute, If the projection is omitted or ‘*’ is used, the query will output all the attributes of input streams.

From all events of the StockExchangeStream stream, output only the events having price >= 20 or price < 100 to the StockQuote stream, where the output event will only have symbol and price as its attributes.

from StockExchangeStream[price >= 20 or price < 100] insert into StockQuote symbol, volume

Here we are only projecting ‘symbol’ and ‘volume’ as the output of the query, and hence the output stream ‘StockQuote’ will only contain ‘symbol’ and ‘volume’ attribute, If the projection is omitted or ‘*’ is used, the query will output all the attributes of input streams.

From all events of the StockExchangeStream stream, output only the events where symbol does not contain 'ws' to the StockQuote stream, where the output event will only have symbol and volume as its attributes.

from StockExchangeStream[not (symbol contains 'ws')] insert into StockQuote symbol, volume

Here we are only projecting ‘symbol’ and ‘volume’ as the output of the query, and hence the output stream ‘StockQuote’ will only contain ‘symbol’ and ‘volume’ attribute, If the projection is omitted or ‘*’ is used, the query will output all the attributes of input streams.

7.1.1.2 Windows

rom <stream-name> {<conditions>}#window.<window-name>(<parameters>) insert [<output-type>] into <stream-name> ({<attribute-name>}| ‘*’|)

Window is a limited subset of events from an event stream. Users can define a window and then use the events on the window for calculations. A window has two types of output: current events and expired events. A window emits current events when a new event arrives. Expired events are emitted whenever an existing event has expired from a window.

There are several types of windows.

1. lengthWindowLength windows - a sliding window that keeps last N events. 2. Time window - a sliding window that keeps events arrived within the last T time period. 3. Time batch window - a time window that processes events in batches. A loop collects the incoming events arrived within last T time period, and outputs them as a batch. 4. Length batch window - a length window that outputs events as a batch only at the nth event

arrival. 5. Time length window (not supported in the current version) - a sliding window that keeps the last N events that arrived within the last T time period. 6. Unique window - keeps only the latest events that are unique according to the given unique attribute. 7. First unique window - keeps the first events that are unique according to the given unique attribute.

Siddhi queries can have three different output types: 'current-events', 'expired-events' and 'all-events'. Users can define these output types by adding the following keywords in between 'insert' and 'into' in the syntax.

1. 'current-events' keyword : The output is only triggered when new events arrive at the window. Notifications will not be given when the expired events trigger the query from the window.
2. 'expired-events' keyword : The query emits output only when the expired events trigger it from the window and not from new events.
3. 'all-events' keyword : The query emits output when it is triggered by both newly-arrived and expired events from the window.
4. No keyword is given : By default, the query assigns 'current-events' to its output stream.

In output event streams, users can define aggregate functions to calculate aggregations within the defined window. CEP supports the following types of aggregate functions.

1. sum 2. avg 3. max 4. min 5. count 6. median (not supported in current version) 7. stddev (not supported in current version) 8. avedev (not supported in current version)

Aggregate function must be named using 'as' keyword. Thus name can be used for referring that attribute, and will be used as the attribute name in the output stream. Following examples shows some queries.

Stream *defined by "**define stream StockExchangeStream (symbol string, price int, volume float);**" is used in below queries.*

Length Window

A sliding window that keeps last N events.

From the events having price >= 20 of the StockExchangeStream stream, output the expiring events of the length window to the StockQuote stream. Here the output events will have symbol and the per symbol average price as their attributes, only if the per symbol average price > 50.

```
from StockExchangeStream[price >= 20]#window.length(50) insert expired-events into StockQuote symbol, avg(price) as avgPrice group by symbol having avgPrice>50
```

In the above query, avg(prize) is an aggregate function.

Time Window

A sliding window that keeps events arrived within the last T time period.

From the events having symbol == 20 of the StockExchangeStream stream, output the both the newly arriving and expiring events of the time window to the IBMStockQuote stream. Here the output events will have maximum, average and minimum prices that has arrived within last minute as their attributes.

```
from StockExchangeStream[symbol == 'IBM']#window.time( 1 min ) insert all-events into IBMStockQuote max(price) as maxPrice, avg(price) as avgPrice, min(price) as minPrice
```

Time Batch Window

A time window that processes events in batches. This in a loop collects the incoming events arrived within last T time period and outputs them as a batch.

From the events of the StockExchangeStream stream, output the events per every 2 minutes from the timeBatch window to the StockQuote stream. Here the output events will have symbol and the per symbol sum of volume for last 2 minutes as their attributes.

from StockExchangeStream#window.timeBatch(2 min) insert into StockQuote symbol, sum(volume) as totalVolume group by symbol

Length Batch Window

A length window that outputs events as a batch only at the nth event arrival.

From the events having price >= 20 of the StockExchangeStream stream, output the expiring events of the lengthBatch window to the StockQuote stream. Here the output events will have symbol and the per symbol average price> 50 as their attributes.

from StockExchangeStream[price >= 20]#window.lengthBatch(50) insert expired-events into StockQuote symbol, avg(price) as avgPrice group by symbol having avgPrice>50

Unique Window

A window that keeps only the latest events that are unique according to the given attribute.

*From the events of the StockExchangeStream stream, output the expiring events of the **unique** window to the StockQuote stream. The output events have symbol, price and volume as their attributes.*

from StockExchangeStream#window.unique("symbol") insert expired-events into StockQuote symbol, price, volume

Info
Here, the output event is the immediate previous event having the same symbol of the current event.

Unique window is mostly used in Join Queries, E.g If you want to get the current stock price of any symbol, you can join the SymbolStream (has an attribute symbol) with a Unique window as follows;

from SymbolStream#window.lenght(1) unidirectional join StockExchangeStream#window.unique("symbol") insert into StockQuote StockExchangeStream.symbol as symbol,StockExchangeStream.price as lastTradedPrice

You can find a sample at http://ushanib.blogspot.com/2013/02/sample-demonstrate-unique-window-and.html

First Unique Window

A window that keeps the first events that are unique according to the given unique attribute

*From the events of the StockExchangeStream stream, output the events of the **firstUnique** window to the StockQuote stream. The output events have symbol, price and volume as their attributes.*

from StockExchangeStream#window.firstUnique("symbol") insert into StockQuote symbol, price, volume

info
Here, the output event is the first event arriving for each symbol.

FirstUnique window is mostly used in Join Queries, E.g If you want to know if a symbol has ever been traded in this StockExchange, you can join the SymbolStream (has an attribute symbol) with a First unique window as follows;

from SymbolStream#window.lenght(1) unidirectional join StockExchangeStream#window.firstUnique("symbol") insert into AvailableSymbolStream StockExchangeStream.symbol as symbol

You can find a sample at http://ushanib.blogspot.com/2013/02/sample-demonstrate-unique-window-and.html

7.1.1.3 Joins

from <stream>#<window> [unidirectional] join <stream>#<window> [unidirectional] [on <condition>] [within <time>] insert [<output-type>] into <stream-name> ({<attribute-name>}| '*')

5. Join takes two streams as the input
6. Each stream must have an associated window
7. It generates the output events composed of one event from each stream
8. With "on <condition>" Siddhi joins only the events that matches the condition
9. With "within <time>", Siddhi joins only the events that are within that time of each other

Following example shows a join query.

Outputs the matching events via JoinStream from last 2000 TickEvent and the NewsEvent that have arrived within 500 msec.

from TickEvent[symbol=='IBM']#window.length(2000) join NewsEvent#window.time(500) insert into JoinStream *

Join can be in multiple forms

1. join - inner join 2. [((left|right|full) outer) | inner] join - only inner join is supported in the current version

When we join two streams, the events arriving at either stream will trigger a joining process. Siddhi also supports a special 'unidirectional' join. Here only one stream (the stream defined with the 'unidirectional' keyword) will trigger the joining process.

Following shows a sample unidirectional join query

When an event arrives at the TickEvent that will be matched with all NewsEvents that have arrived within 500 msec, and if the TickEvent's symbol == NewsEvent's company, the output event will be generated and sent via JoinStream.

from TickEvent[symbol=='IBM']#window.length(2000) as t unidirectional join NewsEvent#window.time(500) as n on t.symbol == n.company insert into JoinStream *

Here 'join' only triggered when events arrives in TickEvent stream. When no projection is given or when '*' is used, the output stream attributes will contain both the input events attributes, and the output attributes will be named as <input-stream-name>_<attribute> to maintain uniqueness.

7.1.1.4 Patterns

rom [every] <stream> -> [every] <stream> ... <stream> within <time> insert into <stream-name> <attribute-name> {<attribute-name>}

1. Pattern processing is based on one or more input streams. 2. Pattern matches events or conditions about events from input streams against a series of happen before/after relationships. 3. The input event streams of the query should be referenced in order to uniquely identify events of those streams. e1=Stream1[prize >= 20] is an example of a reference. 4. Any event in the output stream is a collection of events received from the input streams, and they satisfy the specified pattern. 5. For a pattern, the output attribute should be named using the 'as' keyword, and it will be used as the output attribute name in the output stream.

Following example show a simple pattern query.

If an event arrival at Stream1 with price >= 20 is followed by an event arrival at Stream2 having price >= 1st event's price, an output event will be triggered via StockQuote stream. The event will have two attributes; 1st event's symbol, and 2nd event's price.

from e1=Stream1[price >= 20] -> e2=Stream2[price >= e1.price] insert into StockQuote e1.symbol as symbol, e2.price as price

Every And Within Keywords

Without "every" keyword, the query will only run once. If you have the "every" enclosing a pattern, then the query runs for every occurrence of that pattern. Furthermore, If "within <time>" is used, Siddhi triggers only the patterns where the first and the last events constituting to the pattern have arrived within the given time period. In the following example, a1 and b1 should be within 3000 msec as specified.

For every infoStock event having action == "buy" following an confirmOrder event having command =="OK", the StockExchangeStream event will be matched when its price is > infoStock event's price.

from every (a1 = infoStock[action == "buy"] -> a2 = confirmOrder[command == "OK"]) -> b1 = StockExchangeStream [price > infoStock.price] within 3000 insert into StockQuote a1.action as action, b1.price as price

Logical Operations

You can combine streams in patterns using logical OR and AND. 1. and - occurrence of two events in any order 2. or - occurrence of an event from either of the steams in any order

Following example shows a sample query. It waits till the 'buy' action form both OrderStock1 and OrderStock2 before matching the prices in StockExchangeStream.

For every OrderStock1 event with action == "buy" and OrderStock2 event with action == "buy", the StockExchangeStream will be matched for events having price > 70 followed by events having price > 75.

from every a1 = OrderStock1[action == "buy"] and a2 = OrderStock2[action == "buy"] -> b1 = StockExchangeStream[price > 70] -> b2 = StockExchangeStream[price > 75] insert into StockQuote a1.action as action, b1.price as priceA, b2.price as priceB

Counting Patterns

You can count the number of event occurrences of the same event stream with the minimum and maximum limits. For example, <1:4> means 1 to 4 events, <2:> means 2 or more,

and [3] means exactly 3 events.

For every two or more infoStock events, the StockExchangeStream will be matched for three events having price > 70 followed by one to four events having price > 75.

from every a1 = infoStock[action == "buy"]<2:> -> b1 = StockExchangeStream[price > 70]<3> -> b2 = StockExchangeStream[price > 75]<1:4> insert into StockQuote a1[0].action as action, b1.price as priceA, b2[2].price as priceB

When referring to the resuts events matching the count pattern, square brackets should be used to access a specific occurrence of that event. In the above example, a1[0] will refer the 1st event of the many events that have matched the pattern and arrived via the 'infoStock' stream.

7.1.1.5 Sequences

from <event-regular-expression-of-streams> within <time> insert into <stream-name> <attribute-name> {<attribute-name>}

With patterns, there can be other events in between the events that match the pattern condition. In contrast, sequences must exactly match the sequence of events without any other events in between.

1. Sequence processing uses one or more streams. 2. As input, it takes a sequence of conditions defined in a simple regular expression fashion. 3. The events of the input streams should be assigned names in order to uniquely identify these events when constructing the query projection. 4. It generates the output event stream such that any event in the output stream is a collection of events arrived from the input streams that exactly matches the order defined in the sequence. 5. For a sequence, the output attribute must be named using the 'as' keyword, and it will be used as the output attribute name.

When "within <time>" is used, just like with patterns, Siddhi will output only the events that are within that time of each other.

After one or more occurrence of infoStock event with action == "buy", the query matches StockExchangeStream events with maximum of one event with price between 70 and 75 and one event with price >= 75

from every a1 = infoStock[action == "buy"]+, b1 = StockExchangeStream[price > 70]?, b2 = StockExchangeStream[price >= 75] insert into StockQuote a1[0].action as action, b1.price as priceA, b2.price as priceBJoin

Following Regular Expressions are supported

* Zero or more matches (reluctant). + One or more matches (reluctant). ? Zero or one match (reluctant). or or

Similar to the pattern's count operation, the '*' and '+' regex operators also output many events occurrences. Hence we have to refer these events using square brackets to access a specific occurrence of that event. In the above example, a1[0] refers to the first matching event arrived via the infoStock stream.