02

Complex Event Processing

Notice

- **Author**
  - José Júlio Alferes (jja@fct.unl.pt)
  - João Moura Pires (jmp@fct.unl.pt)

Bibliography

- **Many examples are extracted and adapted from:**
  - Opher Etzion and Peter Niblett. Event Processing in Action. Manning Publications, 2010.
  - Lukasz Golab and Tamer Özsu. Data Stream Management. Morgan and Claypool, 2010.
  - SiddhiQL Guide 3.1
    - https://docs.wso2.com/display/CEP420/Introduction+to+CEP

Table of Contents

Introduction

Limitations of DSMS

- **DSMS is an extension of DBMS**
  - It focus on producing answers which are continuously updated to adapt to constantly changing input contents

Limitations of DSMS

- **DSMS is an extension of DBMS**
  - It focus on producing answers which are continuously updated to adapt to constantly changing input contents
- **Detection and notification of complex patterns of elements involving sequences and ordering are out of scope**

Limitations of DSMS

- **DSMS is an extension of DBMS**
  - It focus on producing answers which are continuously updated to adapt to constantly changing input contents
- **Detection and notification of complex patterns of elements involving sequences and ordering are out of scope**
  - E.g. If this item is present, and this other item appears afterwards, without a certain kind of item appearing in between …

Limitations of DSMS

- **DSMS is an extension of DBMS**
  - It focus on producing answers which are continuously updated to adapt to constantly changing input contents
- **Detection and notification of complex patterns of elements involving sequences and ordering are out of scope**
  - E.g. If this item is present, and this other item appears afterwards, without a certain kind of item appearing in between …
    - Notify of **sensor failure** if a car is detected by the sensor in the beginning of a street, it is detected by the sensor in the end of the street after some reasonable time, and in between it is never detected by the sensor in the middle of the street

Streams versus Events

- **DSMS don’t assume any meaning of the data in the streams**
  - The semantics is on the users/programmers shoulders
  - In fact, data is seen as continuous updates on a schema whose meaning is solely on the database programmer

Streams versus Events

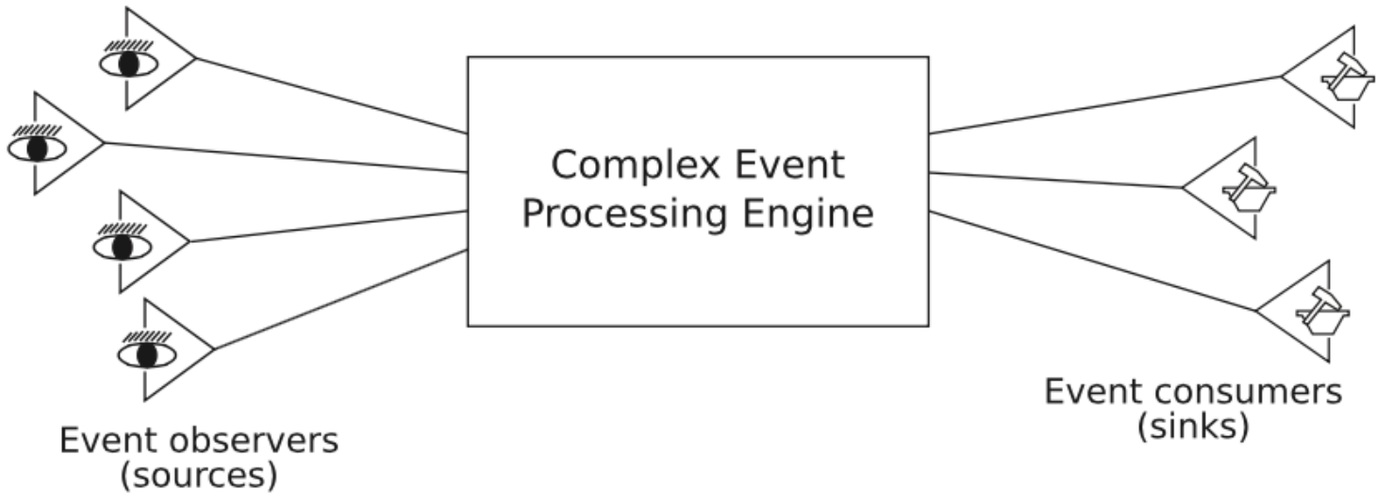
- **DSMS don't assume any meaning of the data in the streams**
  - The semantics is on the users/programmers shoulders
  - In fact, data is seen as continuous updates on a schema whose meaning is solely on the database programmer
- **Complex Event Processor take a very precise meaning of the information that arrives: Events**

Streams versus Events

- **DSMS don’t assume any meaning of the data in the streams**
  - The semantics is on the users/programmers shoulders
  - In fact, data is seen as continuous updates on a schema whose meaning is solely on the database programmer
- **Complex Event Processor take a very precise meaning of the information that arrives: Events**
- **Events are:**
  - notification of things that **happen**;
  - they can happen in the **external world** or in the **system itself**;
  - events are **instantaneous**, and **cannot be deleted** (things don’t “unhappen”!);

Complex Event Processors

- **A CEP engine:**
  - sees event notifications that come from **event observers** (or **event sources**)
  - filters and combines the notifications to understand what is happening in terms of higher-level **composite events**
  - notifies interested clients (**event consumers**) of what is happening in terms of the composite events



Publish-Subscribe

- **CEP has its roots in publish-subscribe messaging patterns**

Publish-Subscribe

- **CEP has its roots in publish-subscribe messaging patterns**
  - message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are

Publish-Subscribe

- **CEP has its roots in publish-subscribe messaging patterns**
  - message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
  - instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to

Publish-Subscribe

- **CEP has its roots in publish-subscribe messaging patterns**
  - message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
  - instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to
  - users **subscribe to classes** of event

Publish-Subscribe

- **CEP has its roots in publish-subscribe messaging patterns**
  - message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
  - instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to
  - users **subscribe to classes** of event
  - the system is responsible for **receiving** the messages from the publishers, **grouping them by classes**, and **delivering** the relevant messages to the relevant subscribers

Composite Events

- **Content-based publish-subscribe system allow subscribers to use complex event filters, based on event content for specifying what they are interested in**

Composite Events

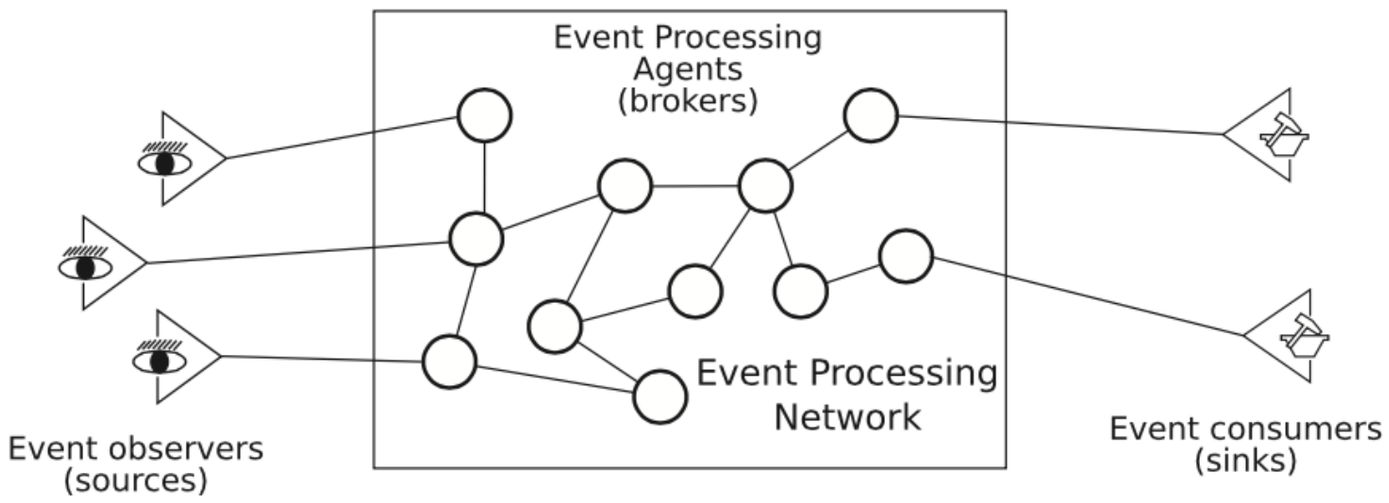
- **Content-based publish-subscribe system allow subscribers to use complex event filters, based on event content for specifying what they are interested in**
- **CEP systems extend this by allowing the filters to depend on the history and on relations between received events**
  - **Composite events** are events derived from the received ones, based on the content, and on the relation to other events already received
    - E.g. a fire is detected (composite derived event) in case three different sensors located in an area smaller than 100m^2 report a temperature higher than 60^{o}C, within 10s from each other

CEP as distributed service

- **CEP focused, from the start, on the features that DSMS lack (even before DSMS appeared!)**
  - But, at least in the beginning, not so much on the features where DSMS excels

CEP as distributed service

- **CEP focused, from the start, on the features that DSMS lack (even before DSMS appeared!)**
  - But, at least in the beginning, not so much on the features where DSMS excels
- **They where mainly thought as a distributed service, able to deal with distributed and heterogeneous information sources**
  - This suggests a distributed architecture with **event brokers** connected in an **event processing network**



Complex Event Processing

Meaning of incoming data

- **In Data Stream Management Systems, the stream of data is not given any particular semantics**
  - Data is just flowing, and the streams are pretty much like tables, but coming in a sequence
  - Sequences are just frequent updates on data

Meaning of incoming data

- **In Data Stream Management Systems, the stream of data is not given any particular semantics**
  - Data is just flowing, and the streams are pretty much like tables, but coming in a sequence
  - Sequences are just frequent updates on data
- **But usually the new data has a very precise meaning!**

Meaning of incoming data

- **In Data Stream Management Systems, the stream of data is not given any particular semantics**
  - Data is just flowing, and the streams are pretty much like tables, but coming in a sequence
  - Sequences are just frequent updates on data
- **But usually the new data has a very precise meaning!**
- **New data comes because something happened inside or outside the system**

Meaning of incoming data

- **In Data Stream Management Systems, the stream of data is not given any particular semantics**
  - Data is just flowing, and the streams are pretty much like tables, but coming in a sequence
  - Sequences are just frequent updates on data
- **But usually the new data has a very precise meaning!**
- **New data comes because something happened inside or outside the system**
- **If we assume (and restrict) new data to be only events, then some things may become easier**

Events

Event: *an **occurrence** within a particular system or domain; it is something that **has happened, or is contemplated as having happened** in that domain. The word event is also used to mean a **programming entity that represents such an occurrence** in a computing system.*

Events

Event: *an **occurrence** within a particular system or domain; it is something that **has happened, or is contemplated as having happened** in that domain. The word event is also used to mean a **programming entity that represents such an occurrence** in a computing system.*

- **Events**
  - are instantaneous;
  - have an associated time when it happened;
  - they indicate something that already happened
    - requests are not events
    - an event cannot be changed or deleted (things don't "unhappen"!)

Complex Event Processing

- **In DSMS data is queried, where data may come in tables or in stream**

Complex Event Processing

- **In DSMS data is queried, where data may come in tables or in stream**
- **In CEP, the name of the game is: *"Given some (raw) events that are detected, derive new (interesting) events"*. E.g.**

Complex Event Processing

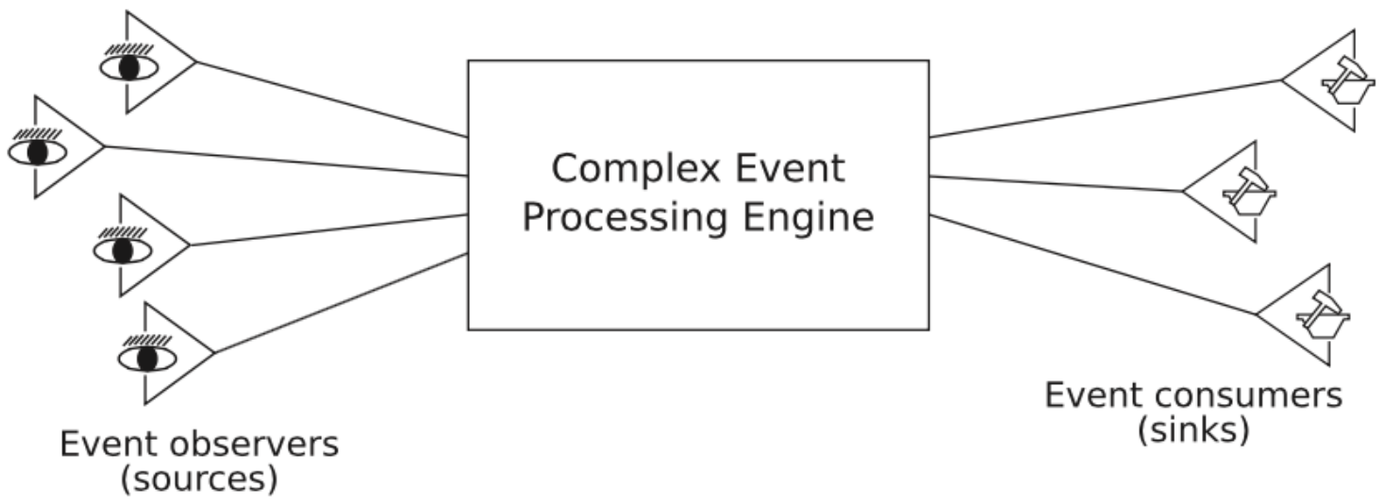
- **In DSMS data is queried, where data may come in tables or in stream**
- **In CEP, the name of the game is: *"Given some (raw) events that are detected, derive new (interesting) events"*. E.g.**
  - whenever one detects the event of a car entering a high way segment, derive an event that a given amount of money is due

Complex Event Processing

- **In DSMS data is queried, where data may come in tables or in stream**
- **In CEP, the name of the game is: *"Given some (raw) events that are detected, derive new (interesting) events"*. E.g.**
  - whenever one detects the event of a car entering a high way segment, derive an event that a given amount of money is due
  - whenever there is an event of a temperature measure above a given values, and of smoke detection, derive the event that there is a fire alarm

Complex Event Processors

- **A CEP engine:**
  - sees event notifications that come from **event observers** (or **event sources**)
  - filters and combines the notifications to understand what is happening in terms of higher-level **composite events**
  - notifies interested clients (**event consumers**) of what is happening in terms of the composite events



Event Producers and Consumers

- **Event producers (sources) may be**
  - sensor readings
  - incoming messages
  - notifications

Event Producers and Consumers

- **Event producers (sources) may be**
  - sensor readings
  - incoming messages
  - notifications
- **Event consumers (sinks) may be**
  - alarms
  - mail messages
  - sms
  - actuators
  - dashboards
  - notifications (including to other CEPs)

Events and Event Types

- **Events are unique occurrences**
  - Computationally these occurrences, when detected, are signalled to a computer system by a given object - an **event object**

Events and Event Types

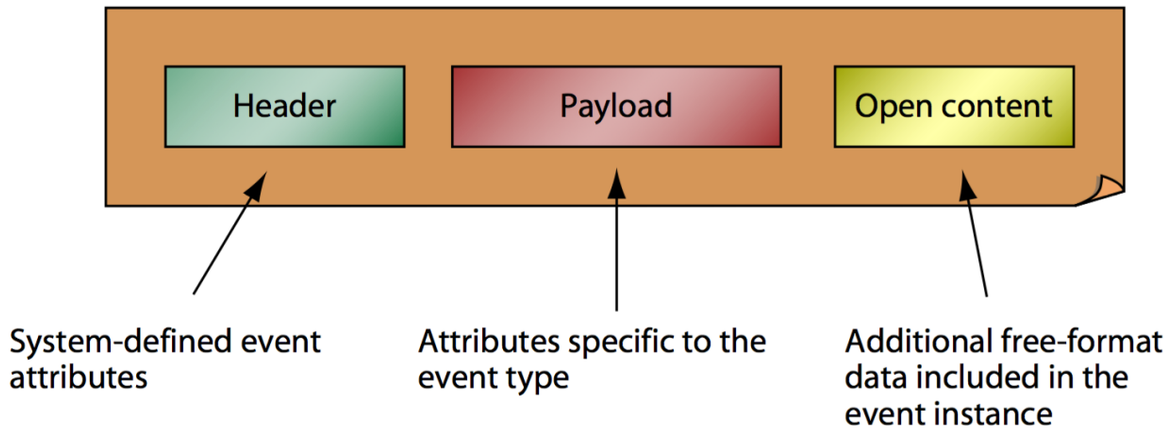
- **Events are unique occurrences**
  - Computationally these occurrences, when detected, are signalled to a computer system by a given object - an **event object**
- **There can be several events that give rise to similar event objects**
  - Its like, in DBs, several tuples with the same schema

Events and Event Types

- **Events are unique occurrences**
  - Computationally these occurrences, when detected, are signalled to a computer system by a given object - an **event object**
- **There can be several events that give rise to similar event objects**
  - Its like, in DBs, several tuples with the same schema

> **Event type:** *specification for a set of* ***event objects that have the same semantic intent and same structure****; every event object is considered to be an instance of an event type*

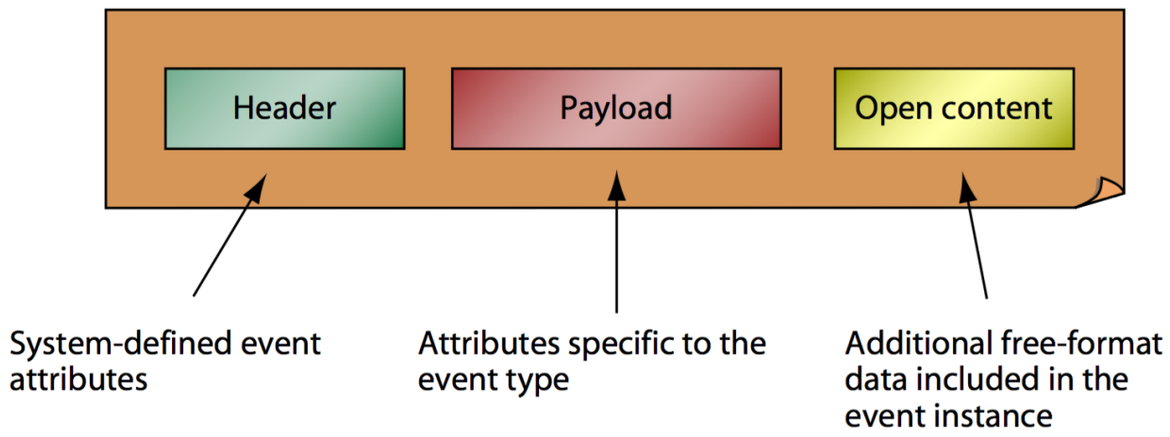
Logical structure of events



Logical structure of events

- **The Header has a system dependent set of attributes. E.g.**
  - type identifier; event id; timestamp; time granularity; occurrence/detection time; source;

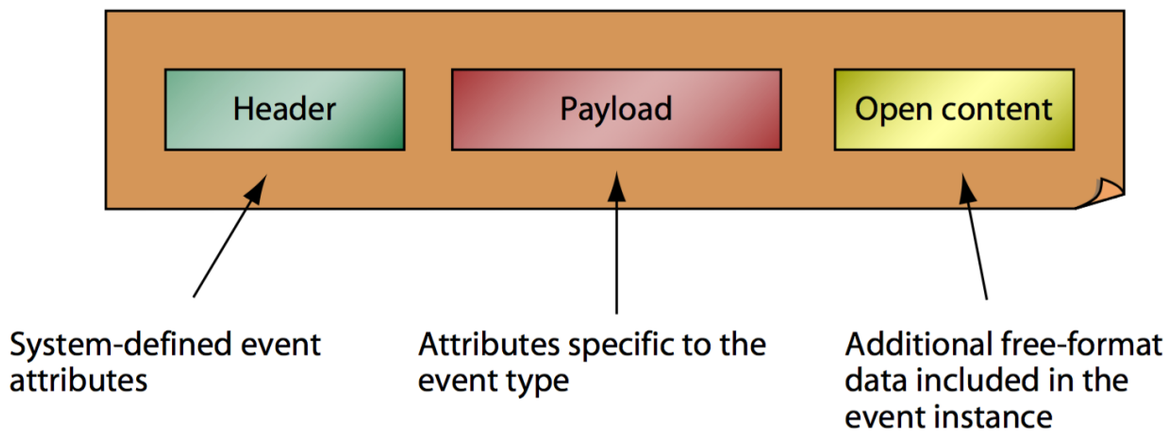
    …



Logical structure of events

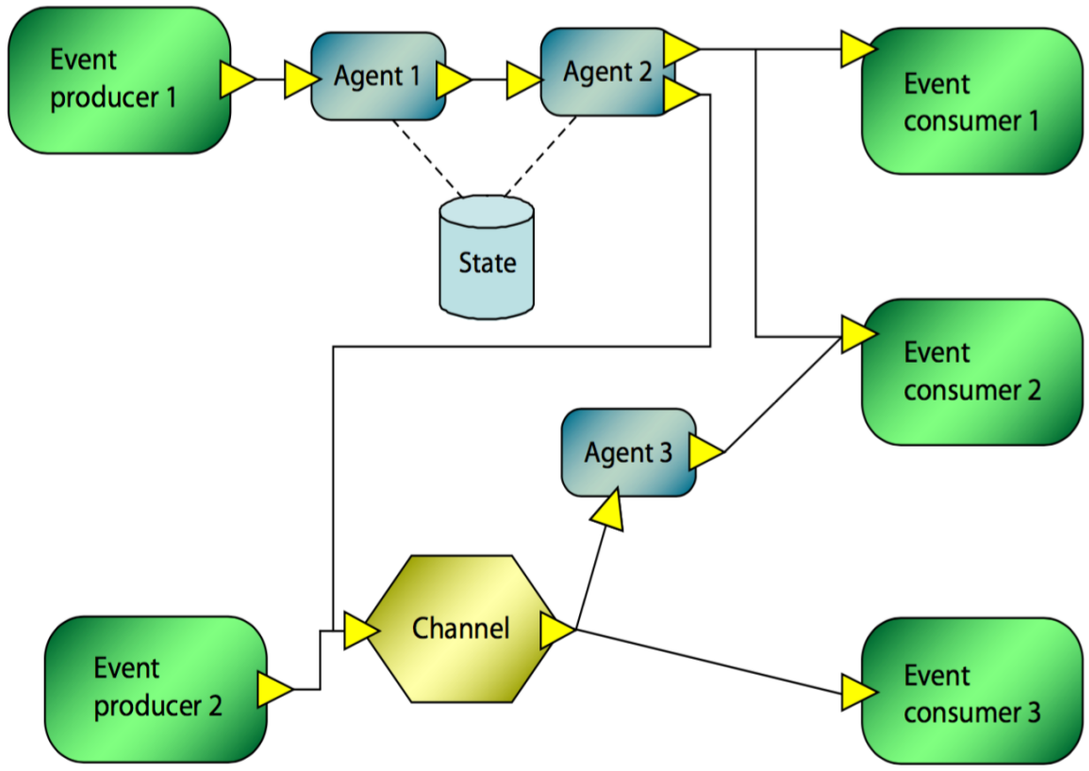
- **The Header has a system dependent set of attributes. E.g.**
  - type identifier; event id; timestamp; time granularity; occurrence/detection time; source;

    …
- **The Payload has a set of attributes that is common to all event object of a given type**
  - It is like in the schema of a table, in a DB



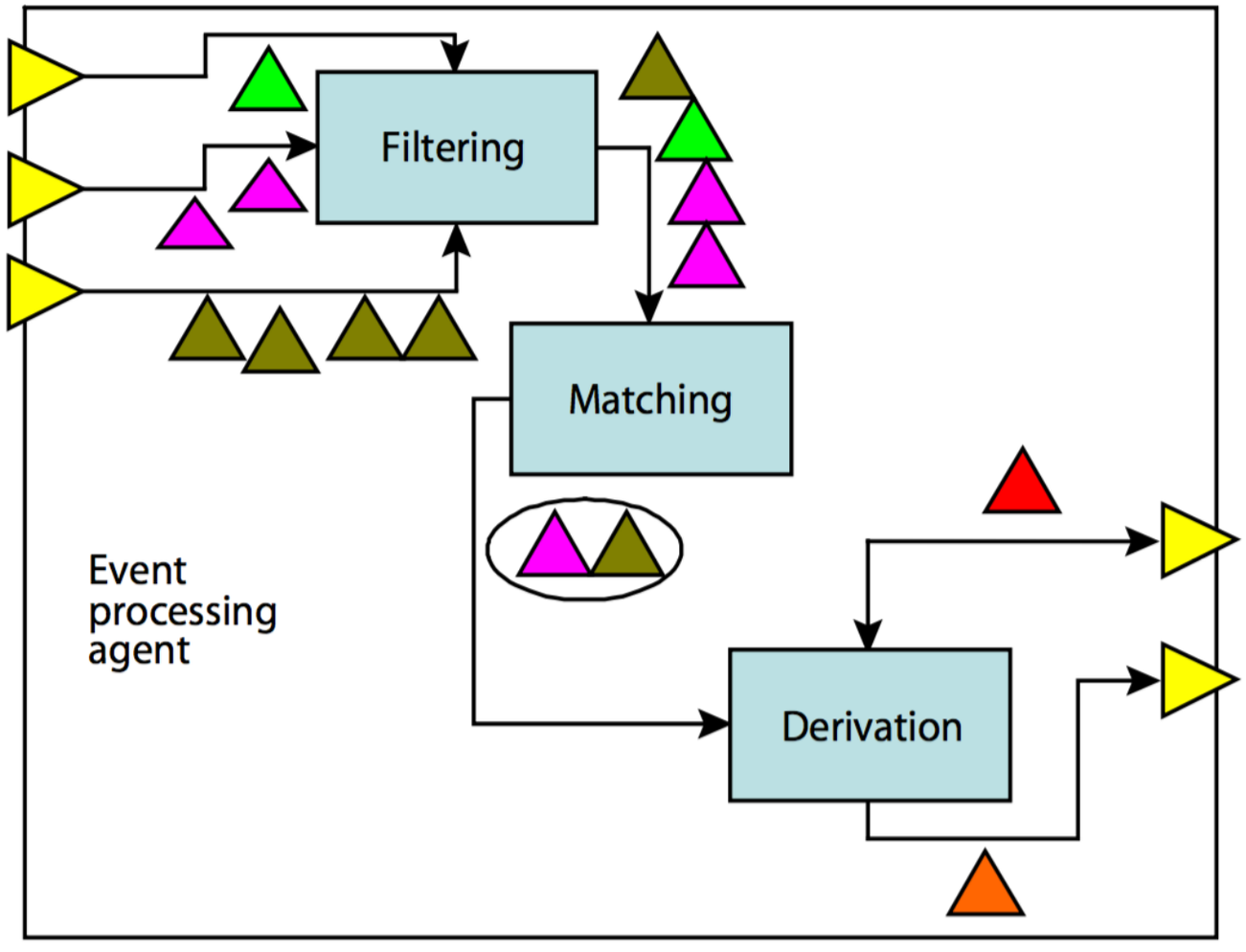
Event Processing Networks

- **It helps to view the process of deriving new events, by several smaller steps**
  - having various event processing agents that take care of the small steps, and connecting them (via channels) in a **network of agents**. E.g.



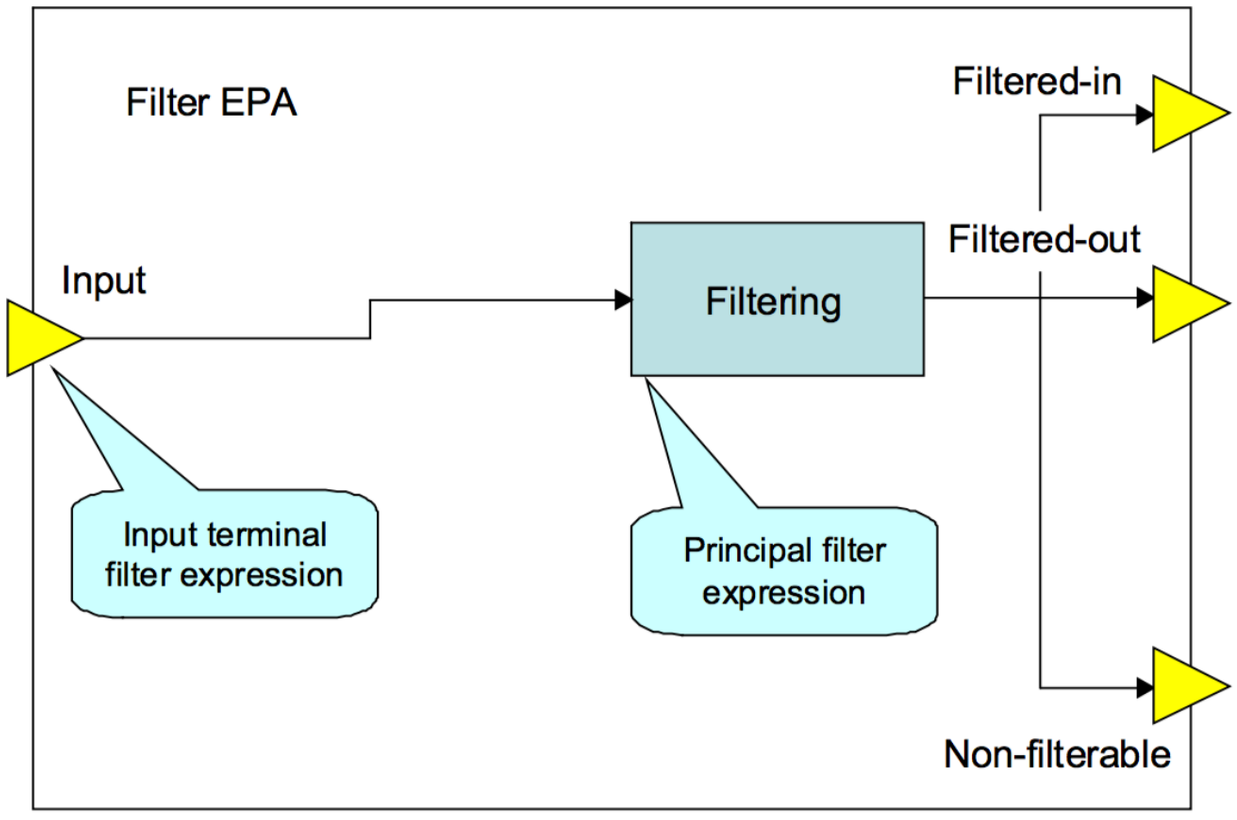
General functions of Agents

- **Filtering:** choosing which input events participate in the processing
- **Matching:** finding/detecting interesting patterns among the events
- **Derivation:** building/deriving new events from the patterns



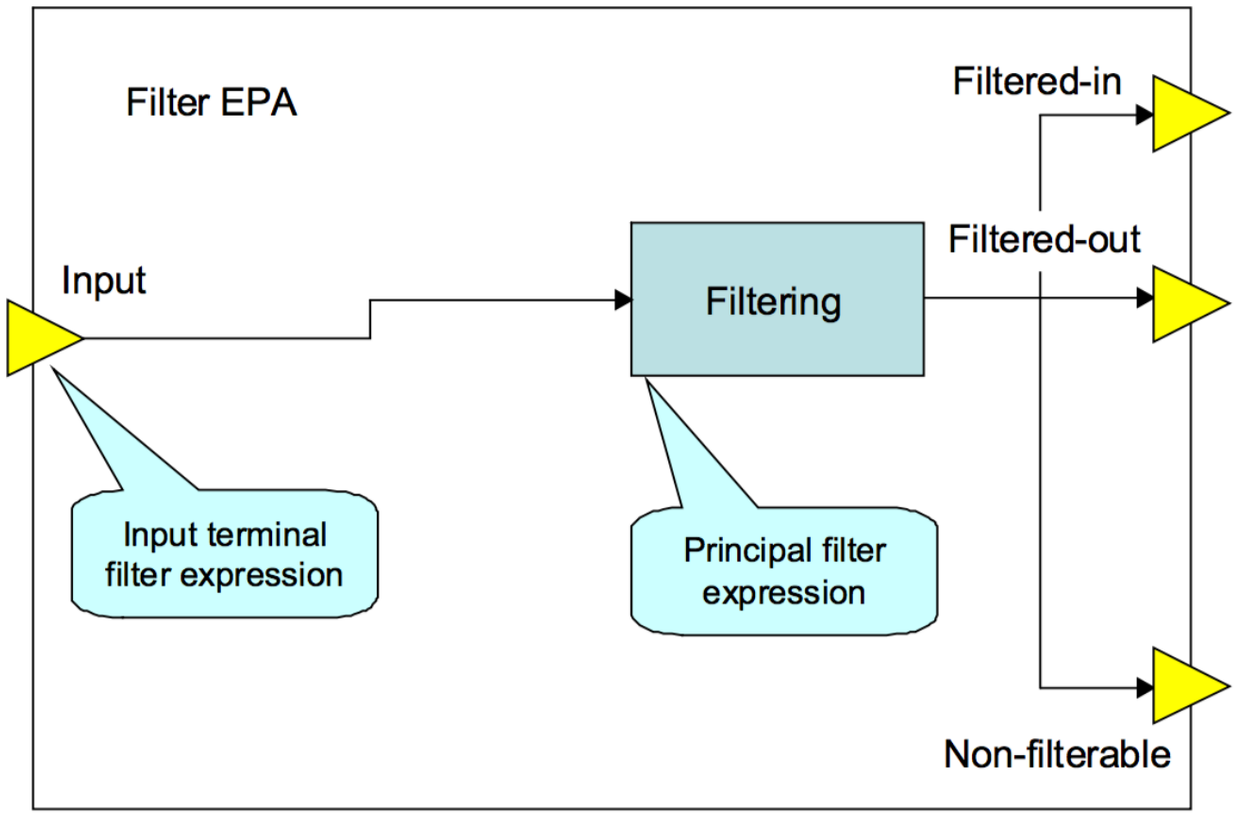
Filtering Events

- **Filtering** can simply put conditions on which events are of interest for processing. E.g.
  - only events of a given type;
  - only event where an attribute has a given value (or is >, or <, or etc); …



Filtering Events

- **Filtering** can simply put conditions on which events are of interest for processing. E.g.
  - only events of a given type;
  - only event where an attribute has a given value (or is >, or <, or etc); …
- **It can also depend on the state. E.g.**
  - only accept every other event;
  - or one event per minute;
  - discard all events that have not arrived in the last 5 minutes; …



Deriving events

- **There may be several types of derivation:**

Deriving events

- **There may be several types of derivation:**
  - **Project**: Pass the event, throwing away some attributes

Deriving events

- **There may be several types of derivation:**
  - **Project**: Pass the event, throwing away some attributes
  - **Translate**: when an event is detected, derive a different event

Deriving events

- **There may be several types of derivation:**
  - **Project**: Pass the event, throwing away some attributes
  - **Translate**: when an event is detected, derive a different event
  - **Enrich**: Given an event, enrich it with computed attributes
    - (e.g. when an event of a car in the highway is detected, enrich it with the number of the segment where the car is)

Deriving events

- **There may be several types of derivation:**
  - **Project**: Pass the event, throwing away some attributes
  - **Translate**: when an event is detected, derive a different event
  - **Enrich**: Given an event, enrich it with computed attributes
    - (e.g. when an event of a car in the highway is detected, enrich it with the number of the segment where the car is)
  - **Splitting**: Split the detected events among several channels

Deriving events

- **There may be several types of derivation:**
  - **Project**: Pass the event, throwing away some attributes
  - **Translate**: when an event is detected, derive a different event
  - **Enrich**: Given an event, enrich it with computed attributes
    - (e.g. when an event of a car in the highway is detected, enrich it with the number of the segment where the car is)
  - **Splitting**: Split the detected events among several channels
  - **Aggregate**: Based on several events that were detected, derive an event with a summary - it depends on past event/state
    - (e.g. every 5 minutes, derive an event with the number of events with different cars that have been received)

Event patterns

- **An event pattern is a template specifying one or more combinations of events**
  - They work as an algebra of events: given a sequence of events, the pattern signals composite/complex events
  - An event pattern is detected relative to a set of events

Event patterns

- **An event pattern is a template specifying one or more combinations of events**
  - They work as an algebra of events: given a sequence of events, the pattern signals composite/complex events
  - An event pattern is detected relative to a set of events
- **Basic patterns: basic operations on events, that do not explicitly need the time in which they happened**

Event patterns

- **An event pattern is a template specifying one or more combinations of events**
  - They work as an algebra of events: given a sequence of events, the pattern signals composite/complex events
  - An event pattern is detected relative to a set of events

- **Basic patterns: basic operations on events, that do not explicitly need the time in which they happened**

- **Time dimension patterns: depend on the history of events**

Basic patterns

- **Conjunction: ALL [list of event types]**
  - It is detected whenever the set has an instance of each of the types in the list
  - E.g. ALL [flight reserved, car reserved, hotel reserved] is detected whenever, in the set, there is an event of both a flight, a car and a hotel reservation (e.g. to detect that everything is ready for the travel)

Basic patterns

- **Conjunction: ALL [list of event types]**
  - It is detected whenever the set has an instance of each of the types in the list
  - E.g. ALL [flight reserved, car reserved, hotel reserved] is detected whenever, in the set, there is an event of both a flight, a car and a hotel reservation (e.g. to detect that everything is ready for the travel)
- **Disjunction: ANY [list of event types]**
  - It is detected whenever the set has an instance of at least one type in the list
  - E.g. ANY [salary paid, loan accepted] is detected whenever either the salary is paid, or a loan is accepted (e.g. to start spending money)

Basic patterns

- **Negation: ABSENCE [event type]**
  - It is detected whenever the set has no instances of the event type
  - This only makes sense when associated with filters that depend on state
    - Suppose we have a filter that only accepts events in the last 5 minutes
    - ABSENCE [sensor measure] is detected whenever no sensor measure is received in the last 5 minutes

Basic patterns

- **Negation: ABSENCE [event type]**
  - It is detected whenever the set has no instances of the event type
  - This only makes sense when associated with filters that depend on state
    - Suppose we have a filter that only accepts events in the last 5 minutes
    - ABSENCE [sensor measure] is detected whenever no sensor measure is received in the last 5 minutes
- **Threshold patterns**
  - Aggregate events in a set, and derives a pattern whenever the aggregated value passes a given threshold
  - Again, it only makes sense with filters that depend on state
    - E.g. whenever the average of the temperature measures in the last 5 minutes is above a certain value (raise an alarm)

Time dimension patterns

- **Sequence: <e_1, e_2, …, e_n>**
  - It is detected whenever all of e_1 through e_n belong to the set, and e_1 occurred before e_2, and … before e_n
  - E.g. with a filter that considers only the events in the last 2 days, <dismissed patient, admitted patient> is detected whenever a patient is admitted less than 2 days after being dismissed

Time dimension patterns

- **Sequence: <e_1, e_2, …, e_n>**
  - It is detected whenever all of e_1 through e_n belong to the set, and e_1 occurred before e_2, and … before e_n
  - E.g. with a filter that considers only the events in the last 2 days, <dismissed patient, admitted patient> is detected whenever a patient is admitted less than 2 days after being dismissed
- **Trend patterns: INCREASING attribute (resp. DECREASING att)**
  - Is detected whenever the value of the attribute in the set is always increasing (resp. decreasing)
  - E.g. with a filter that considers only events of temperature sensors in the last 5 minutes, INCREASING temperature is detected whenever the temperature is strictly increasing for the last 5 minutes

General Patterns

- **All of this is quite general!**

General Patterns

- **All of this is quite general!**
- **These are simply examples of (useful) patterns that can be implemented in a CEP.**
  - Some CEPs have these, some have more, some have less

General Patterns

- **All of this is quite general!**
- **These are simply examples of (useful) patterns that can be implemented in a CEP.**
  - Some CEPs have these, some have more, some have less

- **Maybe it is time to make things a bit more concrete, and focus on the specificities of a particular CEP!**

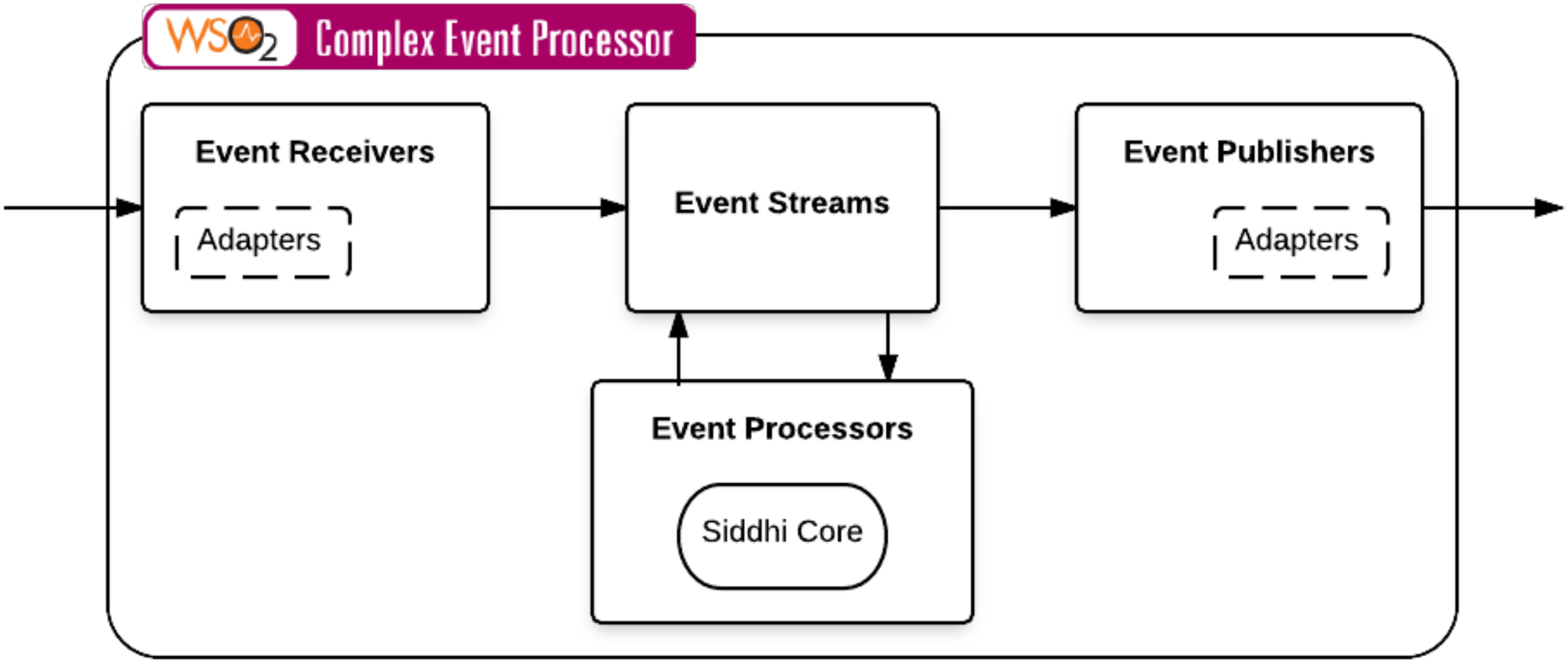
SiddhiQL - query language of WSO2 CEP

WSO2 Complex Event Processor (CEP)

- WSO2 Complex Event Processor (CEP) is a lightweight, easy-to-use, **open source Complex Event Processing server (CEP)**.
  - It identifies the most meaningful events within the event cloud, analyzes their impact, and acts on them in real-time.
  - It's built to be extremely high performing with **WSO2 Siddhi** and massively **scalable using Apache Storm**
  - The CEP can be tightly integrated with **WSO2 Data Analytics Server**, by **adding support for recording and post processing events** with Map-Reduce via Apache Spark, and WSO2 Machine Learner for predictive analytics.

WSO2 Stream Processor (WSO2 SP)

WSO2 - CEP Architecture



SiddhiQL Basics

https://en.wikipedia.org/wiki/WSO2

- **Siddhi QL is the language used by WSO_2 for defining derived composite events**
  - It is called a query language, but it is not exactly for querying
    - (unless you consider querying as only being continuous querying)

SiddhiQL Basics

https://en.wikipedia.org/wiki/WSO2

- **Siddhi QL is the language used by WSO_2 for defining derived composite events**
  - It is called a query language, but it is not exactly for querying
    - (unless you consider querying as only being continuous querying)
- **It is a stream based event definition language**
  - I.e. events come in streams of events
  - In an **event stream all events are of the same type**
  - Event streams come from the outside, or may be defined internally (to use as event channels)

SiddhiQL Basics

https://en.wikipedia.org/wiki/WSO2

- **Siddhi QL is the language used by WSO_2 for defining derived composite events**
  - It is called a query language, but it is not exactly for querying
    - (unless you consider querying as only being continuous querying)
- **It is a stream based event definition language**
  - I.e. events come in streams of events
  - In an **event stream all events are of the same type**
  - Event streams come from the outside, or may be defined internally (to use as event channels)
- **A SiddhiQL "query" is a rule that given one or more event streams, produces an event stream**

Event stream

- **An event stream is an unbound sequence of event objects**

Event stream

- **An event stream is an unbound sequence of event objects**
- **All events in a stream have the same type**
  - Elsewhere in WSO_2, one has to filter the incoming events into the appropriate streams

Event stream

- **An event stream is an unbound sequence of event objects**
- **All events in a stream have the same type**
  - Elsewhere in WSO_2, one has to filter the incoming events into the appropriate streams
- **One can define new (internal) streams with**

```
define stream <name>
 (<attribute> <type>, …, <attribute> <type>);
```

  - **Types** are the ones that you'd expect (see manual)

Event stream

- **An event stream is an unbound sequence of event objects**
- **All events in a stream have the same type**
  - Elsewhere in WSO_2, one has to filter the incoming events into the appropriate streams
- **One can define new (internal) streams with**

```
define stream <name>
 (<attribute> <type>, …, <attribute> <type>);
```

  - **Types** are the ones that you'd expect (see manual)
- **E.g.**

```
define stream TempStream
 (deviceID long, roomNo int, temp double);
```

Basic Event Queries

- **A basic query has the form**

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

- This **defines an agent** (and **an output stream**) that whenever it receives an event in the input, projects the given attributes, and raises an event with the projected attributes to the output stream

Basic Event Queries

- **A basic query has the form**

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

- This **defines an agent** (and **an output stream**) that whenever it receives an event in the input, projects the given attributes, and raises an event with the projected attributes to the output stream

- **E.g.**

```
from TempStream
select roomNo, temp
insert into RoomTempStream;
```

Translate and Enrich

- In a basic query one can also **rename/translate attributes**, or **enrich** by using a number of function similar to SQL in the select clause

Translate and Enrich

- In a basic query one can also **rename/translate attributes**, or **enrich** by using a number of function similar to SQL in the select clause

- **E.g**

```
from RoomTempStream
select roomNo, temp, 'C' as scale
insert into EnrichedRoomTempStream;

from TempStream
select roomNo, (temp * 1.8000 + 32.00) as temp, 'F' as scale
insert into TransformedRoomTempStream;
```

Adding a default value and assigning it to an attribute using as.

produces a stream with the temperature in Celsius and another with it in Fahrenheit

Translate and Enrich

- In a basic query one can also **rename/translate attributes**, or **enrich** by using a number of function similar to SQL in the select clause

- **E.g**

```
from RoomTempStream
select roomNo, temp, 'C' as scale
insert into EnrichedRoomTempStream;

from TempStream
select roomNo, (temp * 1.8000 + 32.00) as temp, 'F' as scale
insert into TransformedRoomTempStream;
```

Adding a default value and assigning it to an attribute using as.

produces a stream with the temperature in Celsius and another with it in Fahrenheit

- **There is a big variety of pre-defined function**
  - The usual arithmetic and logical operators
  - Type conversion functions
  - … (see more at the online manual)
  - And you can define even more functions, in Java

Filtering

- **One can associate filtering conditions to each input stream:**

```
from <input stream> [<filter conditions>]
select <attribute>, …, <attribute>
insert into <output stream name>;
```

Filtering

- **One can associate filtering conditions to each input stream:**

```
from <input stream> [<filter conditions>]
select <attribute>, …, <attribute>
insert into <output stream name>;
```

- **The simplest condition filters are based on logical conditions on the attributes of the stream. E.g.**

```
from TempStream [ roomNo > 245 and
                  roomNo <= 365 and temp > 40 ]
select roomNo, temp
insert into AlertServerRoomTempStream ;
```

Filtering

- **One can associate filtering conditions to each input stream:**

```
from <input stream> [<filter conditions>]
select <attribute>, …, <attribute>
insert into <output stream name>;
```

- **The simplest condition filters are based on logical conditions on the attributes of the stream. E.g.**

```
from TempStream [ roomNo > 245 and
                  roomNo <= 365 and temp > 40 ]
select roomNo, temp
insert into AlertServerRoomTempStream ;
```

- **Expect the usual constructs for the conditions (see manual)**

Window based filtering

- One can also filter incoming events with conditions that depend on time and state

Window based filtering

- One can also filter incoming events with conditions that depend on time and state
- This is done by defining windows in the stream, with:

```
from <input stream> #window.<window type>(<parameters>)
select …
```

Window based filtering

- One can also filter incoming events with conditions that depend on time and state
- This is done by defining windows in the stream, with:

```
from <input stream> #window.<window type>(<parameters>)
select …
```

- Do not confuse these windows with the windows in CQL for s2r operations!!
  - These windows do not transform a stream of data into a relation
  - Rather, **they transform a stream of events, into a sub stream**

Window based filtering

- One can also filter incoming events with conditions that depend on time and state
- This is done by defining windows in the stream, with:

```
from <input stream> #window.<window type>(<parameters>)
select …
```

- Do not confuse these windows with the windows in CQL for s2r operations!!
  - These windows do not transform a stream of data into a relation
  - Rather, **they transform a stream of events, into a sub stream**
- **A window emits two types of events whenever a new event is consumed**:
  - **Current** events and **expired** events
  - A window emits a **current-event** with the **new event that arrived**, whenever one arrives
  - It emits an **expired-event** whenever an **event ceases to belong to the sub stream**

Types of windows

- **Siddhi includes quite a variety of types of windows**

Types of windows

- **Siddhi includes quite a variety of types of windows**
  - `time(<t>)` defines a sliding window holding the events that arrived in the **last `t` time**

Types of windows

- **Siddhi includes quite a variety of types of windows**
  - `time(<t>)` defines a sliding window holding the events that arrived in the **last `t` time**
  - `timeBatch(<t>)` defines a **window that is updated every `t` time**, with the events that arrived in the last t time

Types of windows

- **Siddhi includes quite a variety of types of windows**
  - `time(<t>)` defines a sliding window holding the events that arrived in the **last `t` time**
  - `timeBatch(<t>)` defines a **window that is updated every `t` time**, with the events that arrived in the last t time
  - `length(<n>)` defines a sliding window holding the **last `n` events** that arrived

Types of windows

- **Siddhi includes quite a variety of types of windows**
  - `time(<t>)` defines a sliding window holding the events that arrived in the **last `t` time**
  - `timeBatch(<t>)` defines a **window that is updated every `t` time**, with the events that arrived in the last t time
  - `length(<n>)` defines a sliding window holding the **last `n` events** that arrived
  - `lengthBatch(<n>)` as you'd expect

Types of windows

- **Siddhi includes quite a variety of types of windows**
  - `time(<t>)` defines a sliding window holding the events that arrived in the **last `t` time**
  - `timeBatch(<t>)` defines a **window that is updated every `t` time**, with the events that arrived in the last t time
  - `length(<n>)` defines a sliding window holding the **last `n` events** that arrived
  - `lengthBatch(<n>)` as you'd expect
  - `unique(<att>)` defines a sliding window holding the **last unique events according to attribute `att`**

Types of windows

- **Siddhi includes quite a variety of types of windows**
  - `time(<t>)` defines a sliding window holding the events that arrived in the **last `t` time**
  - `timeBatch(<t>)` defines a **window that is updated every `t` time**, with the events that arrived in the last t time
  - `length(<n>)` defines a sliding window holding the **last `n` events** that arrived
  - `lengthBatch(<n>)` as you'd expect
  - `unique(<att>)` defines a sliding window holding the **last unique events according to attribute `att`**
  - … (see more at the manual)

Windows (from the manual): **time**

time

Syntax	`<event>` **`time`**`(<int|long|time>` **`windowTime`**`)`
Extension Type	Window
Description	A sliding time window that holds events that arrived during the last `windowTime` period at a given time, and gets updated for each event arrival and expiry.
Parameter	• **`windowTime`**: The sliding time period for which the window should hold events.
Return Type	Returns current and expired events.
Examples	• `time(20)` for processing events that arrived within the last 20 milliseconds. • `time(2 min)` for processing events that arrived within the last 2 minutes.

Windows (from the manual): **timeBatch**

timeBatch

Syntax	`<event> timeBatch(<int|long|time> windowTime, <int> startTime)`
Extension Type	Window
Description	A batch (tumbling) time window that holds events that arrive during `windowTime` periods, and gets updated for each `windowTime`.
Parameter	**`windowTime`**: The batch time period for which the window should hold events. **startTime** (Optional): This specifies an offset in milliseconds in order to start the window at a time different to the standard time.
Return Type	Returns current and expired events.
Examples	• `timeBatch(20)` processes events that arrive every 20 milliseconds. • `timeBatch(2 min)` processes events that arrive every 2 minutes. • `timeBatch(10 min, 0)` processes events that arrive every 10 minutes starting from the 0th minute. e.g., If you deploy your window at 08:22 and the first event arrives at 08:26, this event occurs within the time window 08.20 - 08.30. Therefore, this event is emitted at 08.30. • `timeBatch(10 min, 1000*60*5)`processes events that arrive every 10 minutes starting from 5th minute. e.g., If you deploy your window at 08:22 and the first event arrives at 08:26, this event occurs within the time window 08.25 - 08.35. Therefore, this event is emitted at 08.35.

Windows (from the manual): **length**

length

Syntax	`<event>` **`length`**`(<int>` **`windowLength`**`)`
Extension Type	Window
Description	A sliding length window that holds the last `windowLength` events at a given time, and gets updated for each arrival and expiry.
Parameter	• **`windowLength`**: The number of events that should be included in a sliding length window.
Return Type	Returns current and expired events.
Examples	• `length(10)` for processing the last 10 events. • `length(200)` for processing the last 200 events.

Windows (from the manual): **lengthBatch**

lengthBatch

Syntax	`<event> lengthBatch(<int> windowLength)`
Extension Type	Window
Description	A batch (tumbling) length window that holds a number of events specified as the `windowLength`. The window is updated each time a batch of events that equals the number specified as the `windowLength` arrives.
Parameter	**`windowLength`**: The number of events the window should tumble.
Return Type	Returns current and expired events.
Examples	• `lengthBatch(10)` for processing 10 events as a batch. • `lengthBatch(200)` for processing 200 events as a batch.

Windows (from the manual): **externalTime**

externalTime

Syntax	`<event> externalTime(<long> timestamp, <int|long|time> windowTime)`
Extension Type	Window
Description	A sliding time window based on external time. It holds events that arrived during the last `windowTime` period from the external timestamp, and gets updated on every monotonically increasing timestamp.
Parameter	• **`windowTime`**: The sliding time period for which the window should hold events.
Return Type	Returns current and expired events.
Examples	• `externalTime(eventTime,20)` for processing events arrived within the last 20 milliseconds from the `eventTime` • `externalTime(eventTimestamp, 2 min)` for processing events arrived within the last 2 minutes from the `eventTimestamp`

Windows (from the manual)

Inbuilt Windows

Following are the supported inbuilt windows of Siddhi

- time
- timeBatch
- length
- lengthBatch
- externalTime
- cron
- firstUnique
- unique
- sort
- frequent
- lossyFrequent
- externalTimeBatch
- timeLength
- uniqueExternalTimeBatch

Windows' output

- **As the output of a windows, one can use either the current events, the expired events, or both**
  - This is signaled by using `current` **`events`**, **`expired`** `events` or **`all`** after the `insert`. E.g:

```
from TempStream#window.time(1 min)
select *
insert expired events into DelayedTempStream;
```

Windows' output

- **As the output of a windows, one can use either the current events, the expired events, or both**
  - This is signaled by using `current` **`events`**, **`expired`** `events` or **`all`** after the `insert`. E.g:

```
from TempStream#window.time(1 min)
select *
insert expired events into DelayedTempStream;
```

    - this rule delays the events in the temperature stream by 1 minute

Windows' output

- **As the output of a windows, one can use either the current events, the expired events, or both**
  - This is signaled by using `current` **`events`**, **`expired`** `events` or **`all`** after the `insert`. E.g:

```
from TempStream#window.time(1 min)
select *
insert expired events into DelayedTempStream;
```

    - this rule delays the events in the temperature stream by 1 minute

- **The use of windows is especially relevant for patterns that aggregate values**

Aggregation over windows

- **One can use usual SQL aggregation functions, group by, and having, with the expected meaning. E.g.:**
  - **Upon all event arrival and expire**, notify of the average temperature of devices over the last minute:

```
from TempStream#window.time(1 min)
select avg(temp) as avgTemp, roomNo, deviceID
insert all into AvgTempStream;
```

Aggregation over windows

- **One can use usual SQL aggregation functions, group by, and having, with the expected meaning. E.g.:**
  - **Upon all event arrival and expire**, notify of the average temperature of devices over the last minute:

```
from TempStream#window.time(1 min)
select avg(temp) as avgTemp, roomNo, deviceID
insert all into AvgTempStream;
```

  - Notify the average temperature per room for the last 10 minutes, if this average is > 40

```
from TempStream#window.time(10 min)
select avg(temp) as avgTemp, roomNo
group by roomNo
having avgTemp > 40
insert current events into AlarmTempStream;
```

Aggregation over windows

- **One can use usual SQL aggregation functions, group by, and having, with the expected meaning. E.g.:**
  - **Upon all event arrival and expire**, notify of the average temperature of devices over the last minute:

```
from TempStream#window.time(1 min)
select avg(temp) as avgTemp, roomNo, deviceID
insert all into AvgTempStream;
```

  - Notify the average temperature per room for the last 10 minutes, if this average is > 40

```
from TempStream#window.time(10 min)
select avg(temp) as avgTemp, roomNo
group by roomNo
having avgTemp > 40
insert current events into AlarmTempStream;
```

  - the “current events” can be omitted (it is assumed by default)

Aggregation over windows: Manual

Following are some inbuilt aggregation functions shipped with Siddhi, for more aggregation functions, see execution extensions.

- avg
- sum
- max
- min
- count
- distinctCount
- maxForever
- minForever
- stdDev

Joining streams

- **One can perform the usual join operation in streams**
  - But this is **only possible for streams with a bound**, defined by a window!

```
from <stream>#<window> join <stream>#<window>
   on <join condition>
   within <time gap>                                    (Optional!)
select …
```

  - Matches all events in the first stream with all events in the second, according to the join condition
    - **if `within` is used, then the joined events must be at most `<time gap>` apart**

Joining streams

- **One can perform the usual join operation in streams**
  - But this is **only possible for streams with a bound**, defined by a window!

```
from <stream>#<window> join <stream>#<window>
   on <join condition>
   within <time gap>                                  (Optional!)
select …
```

  - Matches all events in the first stream with all events in the second, according to the join condition
    - **if `within` is used, then the joined events must be at most `<time gap>` apart**
  - **The event resulting from a join is produced whenever it is possible to join the two events, i.e. at the time of the latest of the two events**

Join example

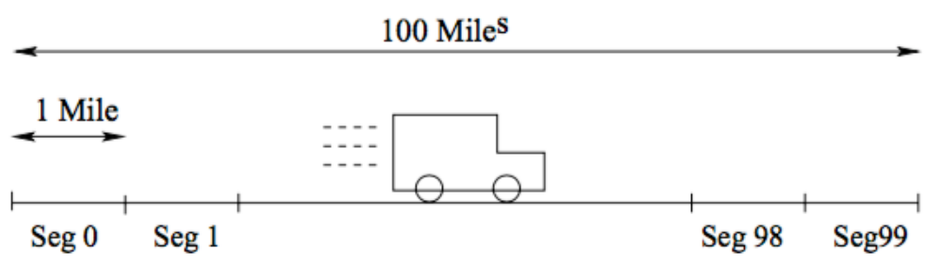
- **Emit alerts to switching on temperature regulator, if they are not already on, for a room which has the temperature above 40 degrees**

Join example

- **Emit alerts to switching on temperature regulator, if they are not already on, for a room which has the temperature above 40 degrees**

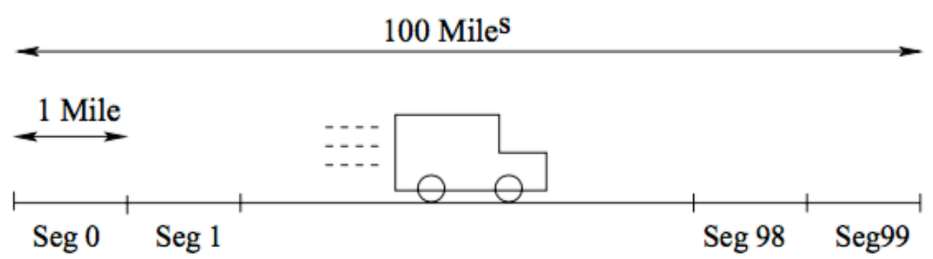
```
define stream RegulatorStream
  (regulatorID long, roomNo int, isOn bool);


from TempStream[temp > 40]#window.time(1 min) as T
     join RegulatorStream[not isOn]#window.length(1) as R
     on T.roomNo == R.roomNo
select T.roomNo, R.regulatorID, 'start' as action
insert into RegulatorActionStream
```

Again the Linear Road



- **The road is divided into 100 segments (of 1Km each)**
- **The input stream has tuples with carID, position, and speed**
  - The position given as the distance in meters from the beginning of the road
- **The output stream has tuples with carID and price to pay for a segment, any time a car enters that segment**
- **The total price to pay for a segment is as follows:**
  - a non-congested segment pays nothing
  - a congested segment pays $2 \times (numCars-50)^2$, where *numCars* is the number of cars in the segment at the moment
  - a segment is congested if the average speed of all cars in the last 5 minutes is less than 40 Km/h

Again the Linear Road



- **The road is divided into 100 segments (of 1Km each)**
- **The input stream has tuples with carID, position, and speed**
  - The position given as the distance in meters from the beginning of the road
- **The output stream has tuples with carID and price to pay for a segment, any time a car enters that segment**
- **The total price to pay for a segment is as follows:**
  - a non-congested segment pays nothing
  - a congested segment pays $2 \times (numCars-50)^2$, where *numCars* is the number of cars in the segment at the moment
  - a segment is congested if the average speed of all cars in the last 5 minutes is less than 40 Km/h

Lets implement it, this time in SiddhiQL!

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;
```

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr                          PosSpeedStr: (carID, position, speed)
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;
```

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr                          PosSpeedStr: (carID, position, speed)
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;                  SegSpeedStr: (carID, speed, segNo)
```

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr                          PosSpeedStr: (carID, position, speed)
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;                  SegSpeedStr: (carID, speed, segNo)
```

- Define the stream with the events of cars entering the segments

```
from SegSpeedStr#window.time(30 s)
select carID, segNo
insert into CarSegEntryStr;
```

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr                          PosSpeedStr: (carID, position, speed)
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;                  SegSpeedStr: (carID, speed, segNo)
```

- Define the stream with the events of cars entering the segments

```
from SegSpeedStr#window.time(30 s)
select carID, segNo
insert into CarSegEntryStr;               CarSegEntryStr: (carID, segNo)
```

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr                          PosSpeedStr: (carID, position, speed)
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;                  SegSpeedStr: (carID, speed, segNo)
```

- Define the stream with the events of cars entering the segments

```
from SegSpeedStr#window.time(30 s)
select carID, segNo
insert into CarSegEntryStr;               CarSegEntryStr: (carID, segNo)
```

- A stream `CarNumStream` with the number of cars in a segment (every 30s)

```
from SegSpeedStr#window.time(30 s)
select segNo, count(carID)
group by segNo
insert into CarNumStream;
```

Linear Road in SiddhiQL

- Start by defining a stream `SegSpeedStr` that in the input stream replaces the position by the number of the segment

```
from PosSpeedStr                          PosSpeedStr: (carID, position, speed)
select carID, speed, position/1000 as segNo
insert into SegSpeedStr;                  SegSpeedStr: (carID, speed, segNo)
```

- Define the stream with the events of cars entering the segments

```
from SegSpeedStr#window.time(30 s)
select carID, segNo
insert into CarSegEntryStr;               CarSegEntryStr: (carID, segNo)
```

- A stream `CarNumStream` with the number of cars in a segment (every 30s)

```
from SegSpeedStr#window.time(30 s)
select segNo, count(carID)
group by segNo
insert into CarNumStream;                 CarNumStream: (segNo, count)
```

Linear Road in SiddhiQL

- **`CongSegStr`** as the stream signalling, at any time, the numbers of congested segments

`SegSpeedStr: (carID, speed, segNo)`

```
from SegSpeedStr#window.time(5 min)
select segNo
group by segNo
having avg(speed) < 40
insert into CongSegStr;
```

Linear Road in SiddhiQL

- **`CongSegStr`** as the stream signalling, at any time, the numbers of congested segments

`SegSpeedStr: (carID, speed, segNo)`

```
from SegSpeedStr#window.time(5 min)
select segNo
group by segNo
having avg(speed) < 40
insert into CongSegStr;
```

`CongSegStr: (segNo)`

Linear Road in SiddhiQL

- **`CongSegStr`** as the stream signalling, at any time, the numbers of congested segments

`SegSpeedStr: (carID, speed, segNo)`

```
from SegSpeedStr#window.time(5 min)
select segNo
group by segNo
having avg(speed) < 40
insert into CongSegStr;
```

`CongSegStr: (segNo)`

- **`TollChargeStr`** as the relation with the current number of cars in each segment

```
from CarSegEntryStr
  join CongSegStr on (CarSegEntryStr.segNo == CongSegStr.segNo)
  join CarNumStream on (CarSegEntryStr.segNo == CarNumStream.segNo)
select CarSegEntryStr.carID, 2*(CarNumStream.numCars-50)**2 as Toll
insert into TollChargeStr;
```

`TollChargeStr: (carID, Toll)`

Linear road variant

- Signalling the toll upon exit, rather than entry, in the segment is quite easy!

Linear road variant

- Signalling the toll upon exit, rather than entry, in the segment is quite easy!
- Just change `CarSegEntry` into (**possibly also changing its name**, to match the new meaning):

- All the rest remains the same!

```
from SegSpeedStr#window.time(30 s)
select carID, segNo
insert expired events into CarSegEntryStr;
```