

02

Complex Event Processing

■ Author

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

- This material can be freely used for personal or academic purposes without any previous authorization from the author, provided that this notice is maintained/kept.
- For commercial purposes the use of any part of this material requires the previous authorization from the author(s).

Bibliography

- **Many examples are extracted and adapted from:**

- ◆ Opher Etzion and Peter Niblett. Event Processing in Action. Manning Publications, 2010.
- ◆ Lukasz Golab and Tamer Özsü. Data Stream Management. Morgan and Claypool, 2010.
- ◆ SiddhiQL Guide 3.1
 - <https://docs.wso2.com/display/CEP420/Introduction+to+CEP>

Table of Contents

- **Introduction**
- **Complex Event Processing**
- **SiddhiQL - query language of WSO2 CEP**
- **Other approaches to CEP**

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**

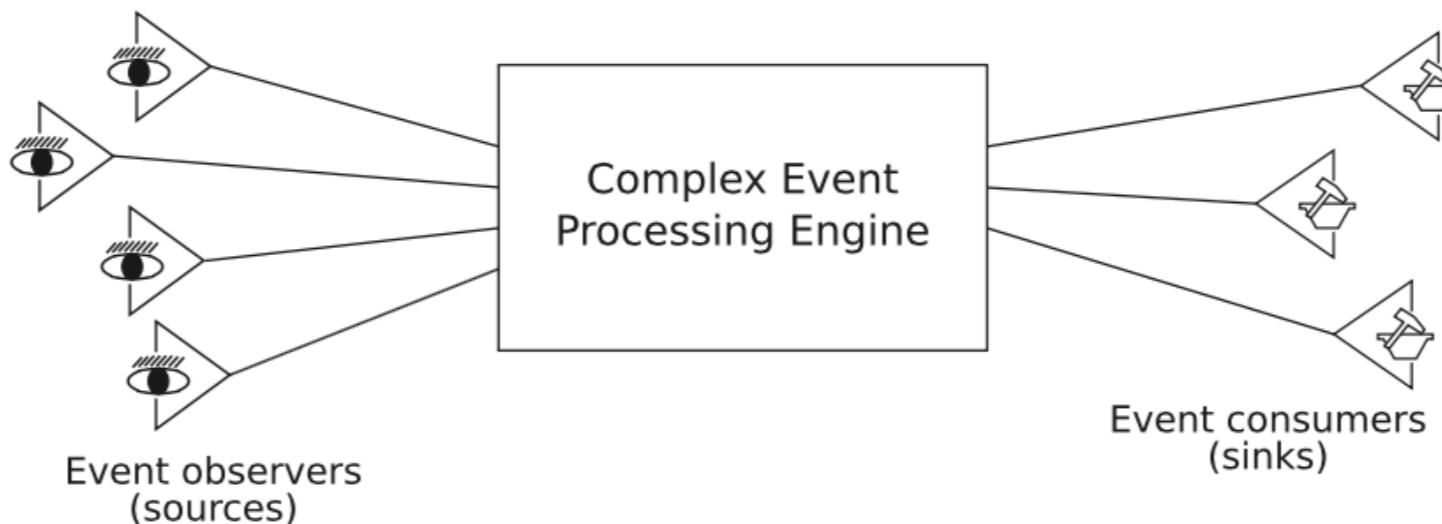
Streams versus Events

- **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

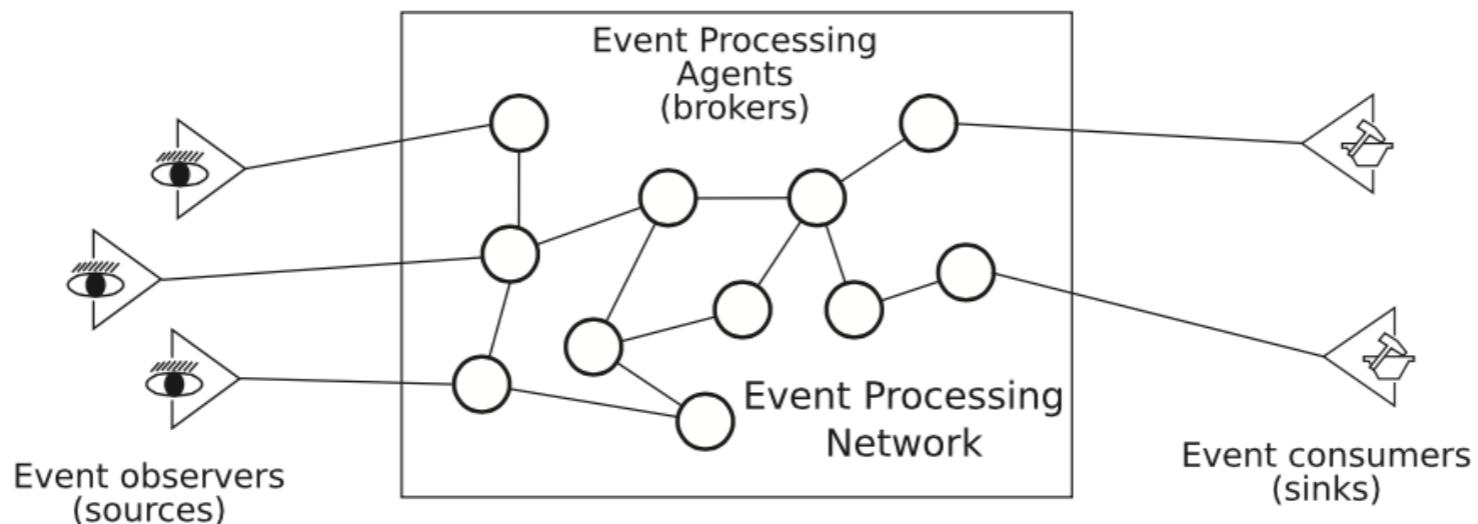
- Content-based publish-subscribe system allow subscribers to use complex event filters, based on event content for specifying what they are interested in
- CEP systems extend this by allowing the filters to depend on the history and on relations between received events
 - ◆ Composite events are events 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 100m² report a temperature higher than 60°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 were 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

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.

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

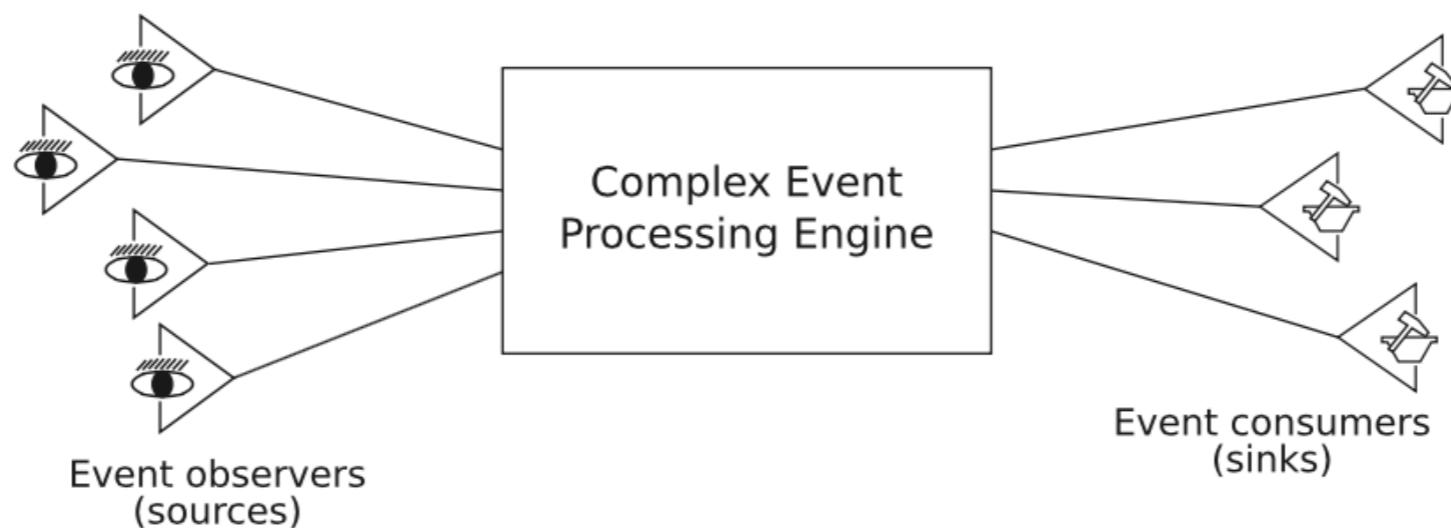
Complex Event Processing

- 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

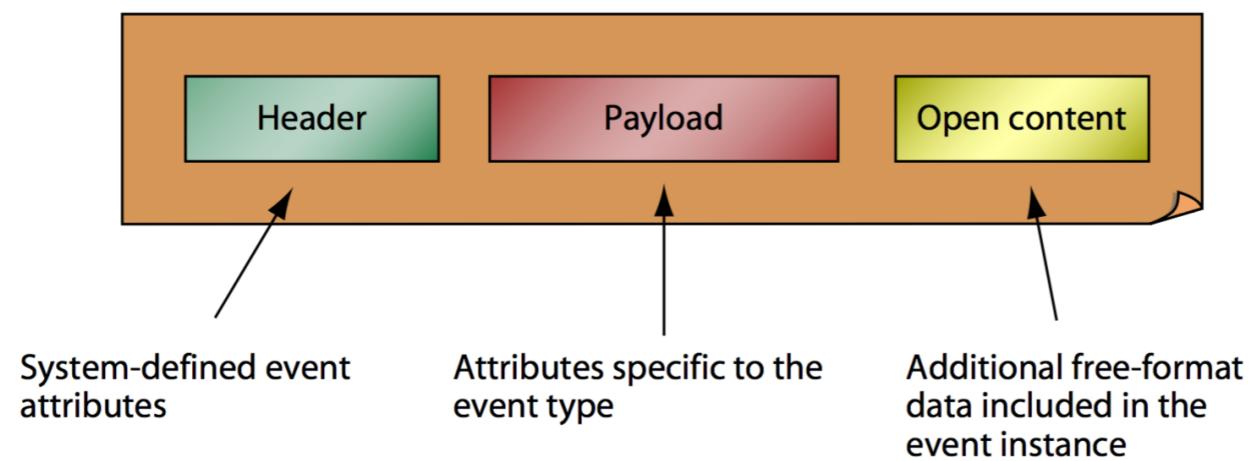
- ◆ Computationally these occurrences, when detected, are signalled to a computer system by a given object - an **event object**

■ 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

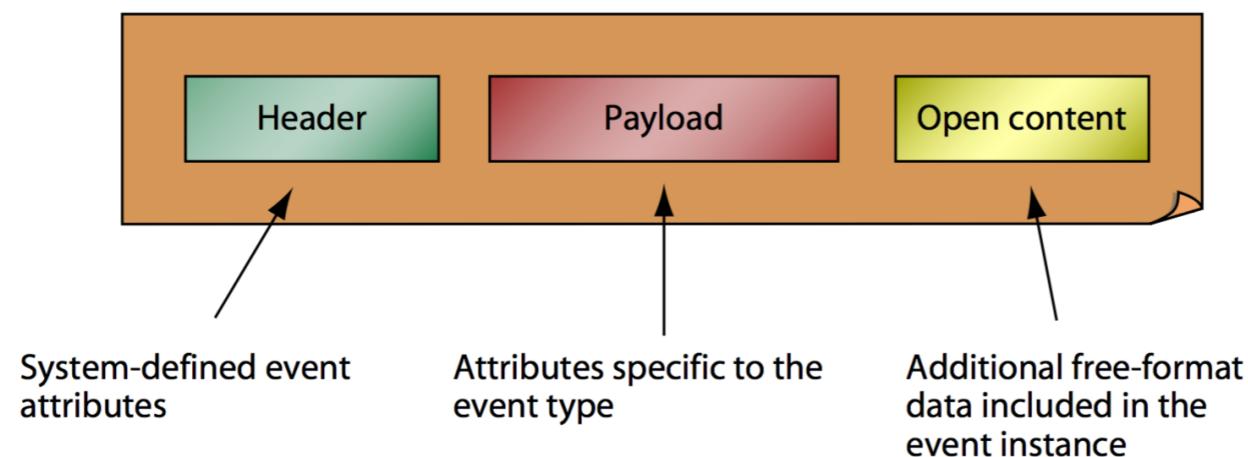


Logical structure of events

■ The Header has a system dependent set of attributes. E.g.

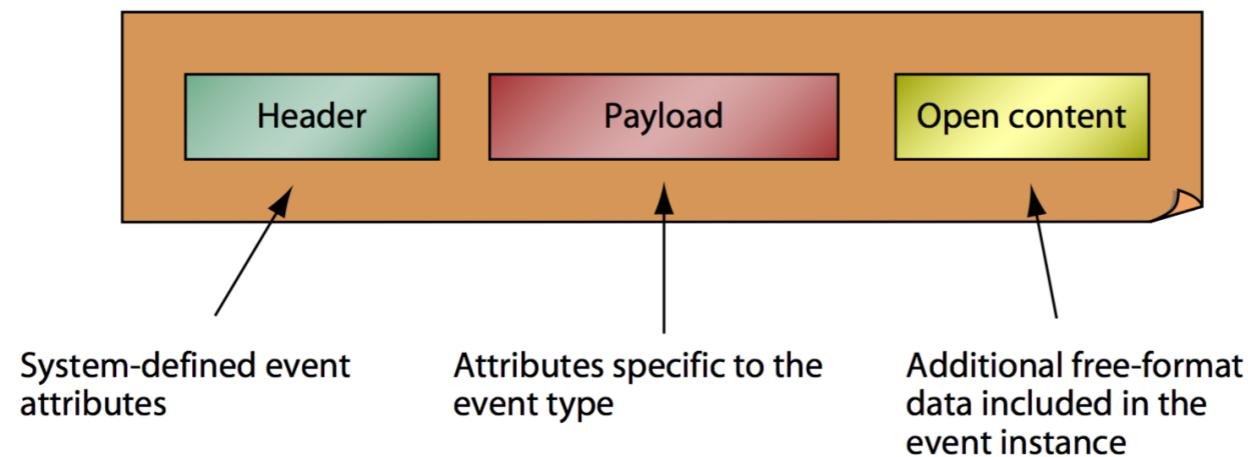
- ◆ type identifier; event id; timestamp; time granularity; occurrence/detection time; source;

...



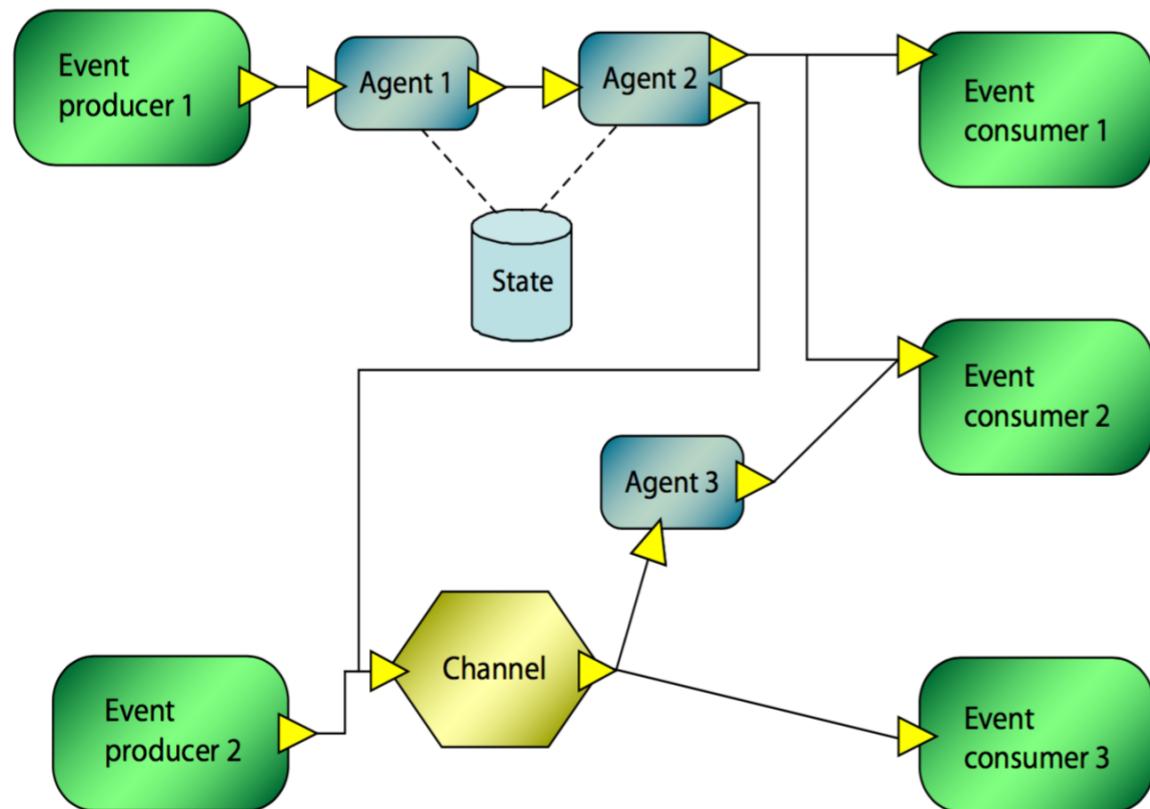
Logical structure of events

- 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



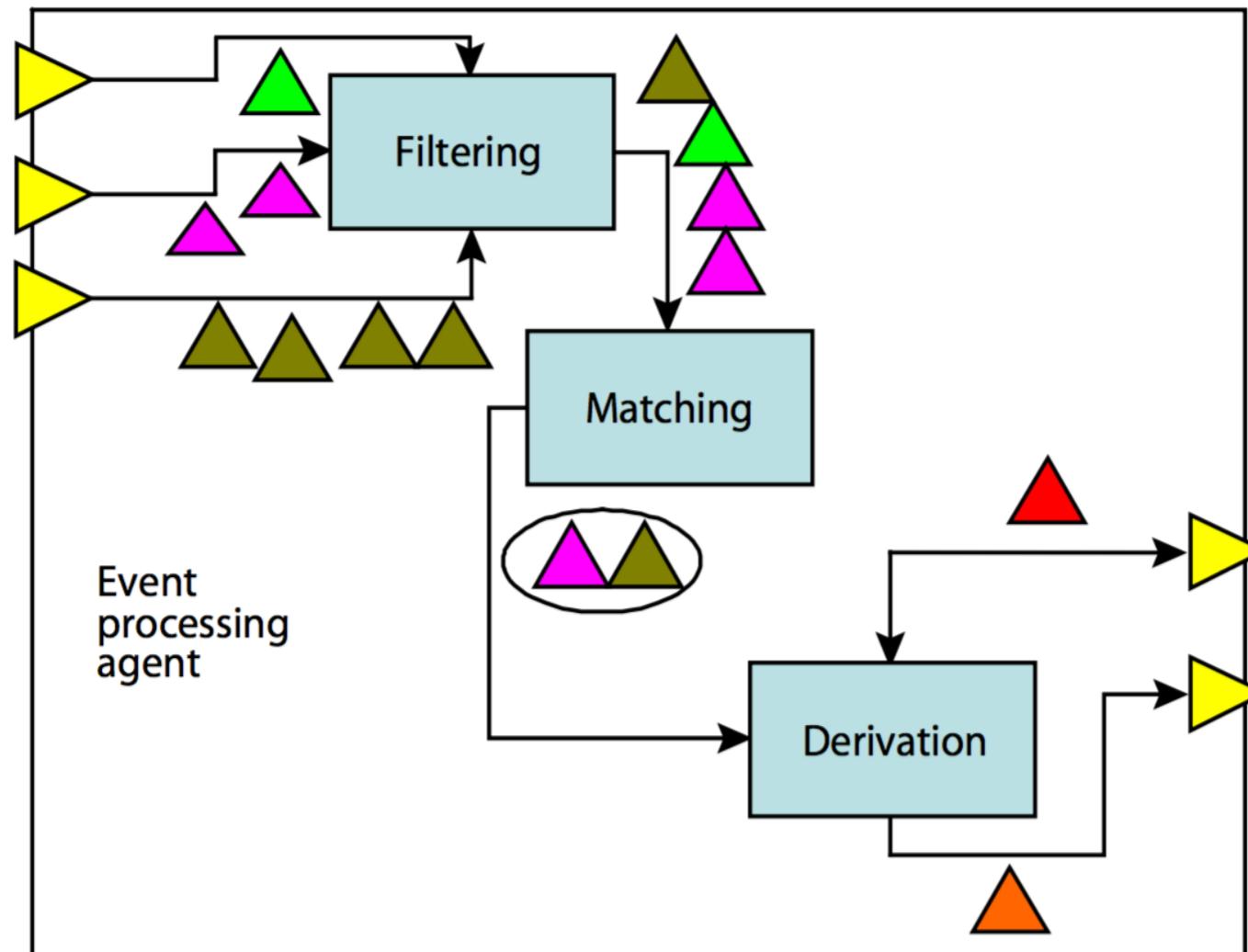
Event Processing Networks

- 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.



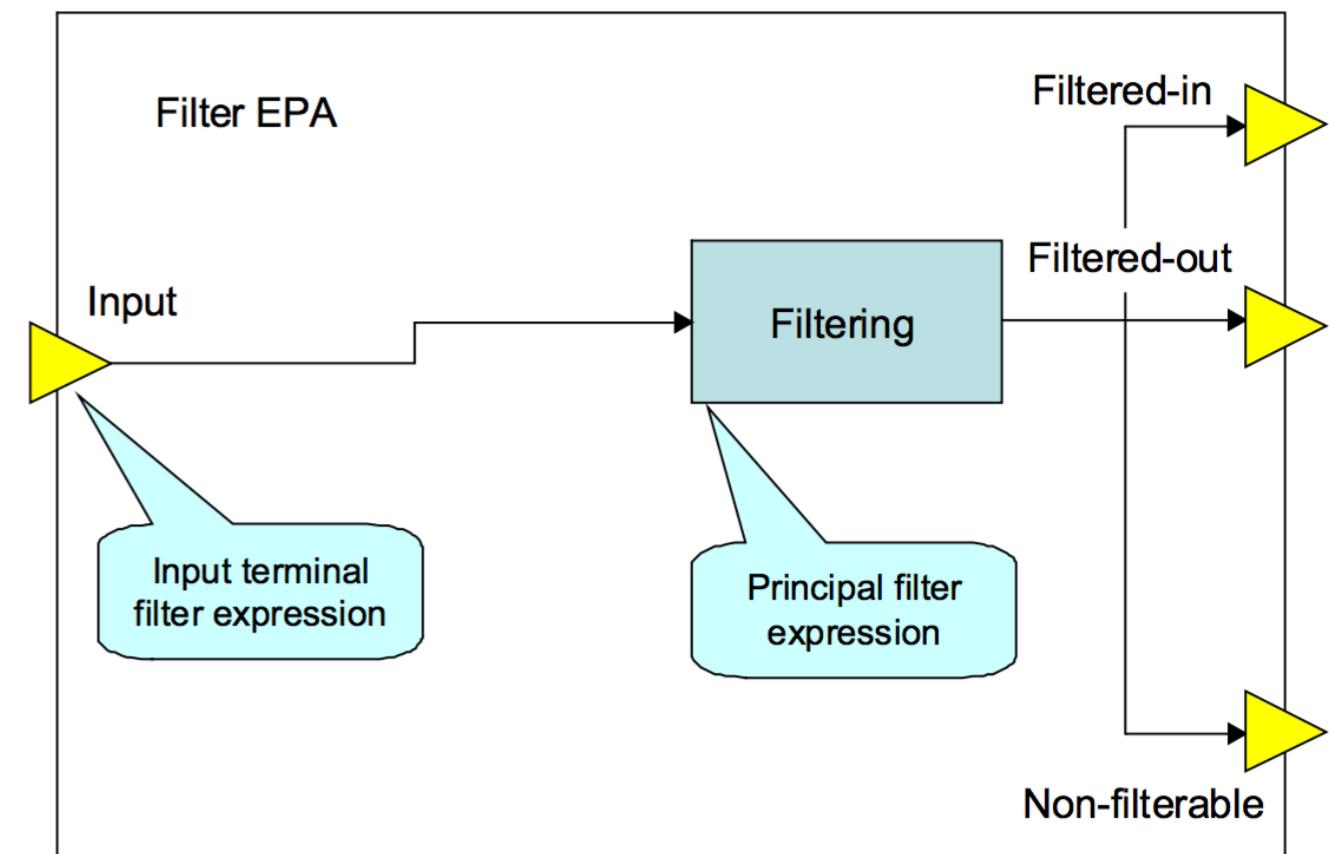
General functions of Agents

- **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

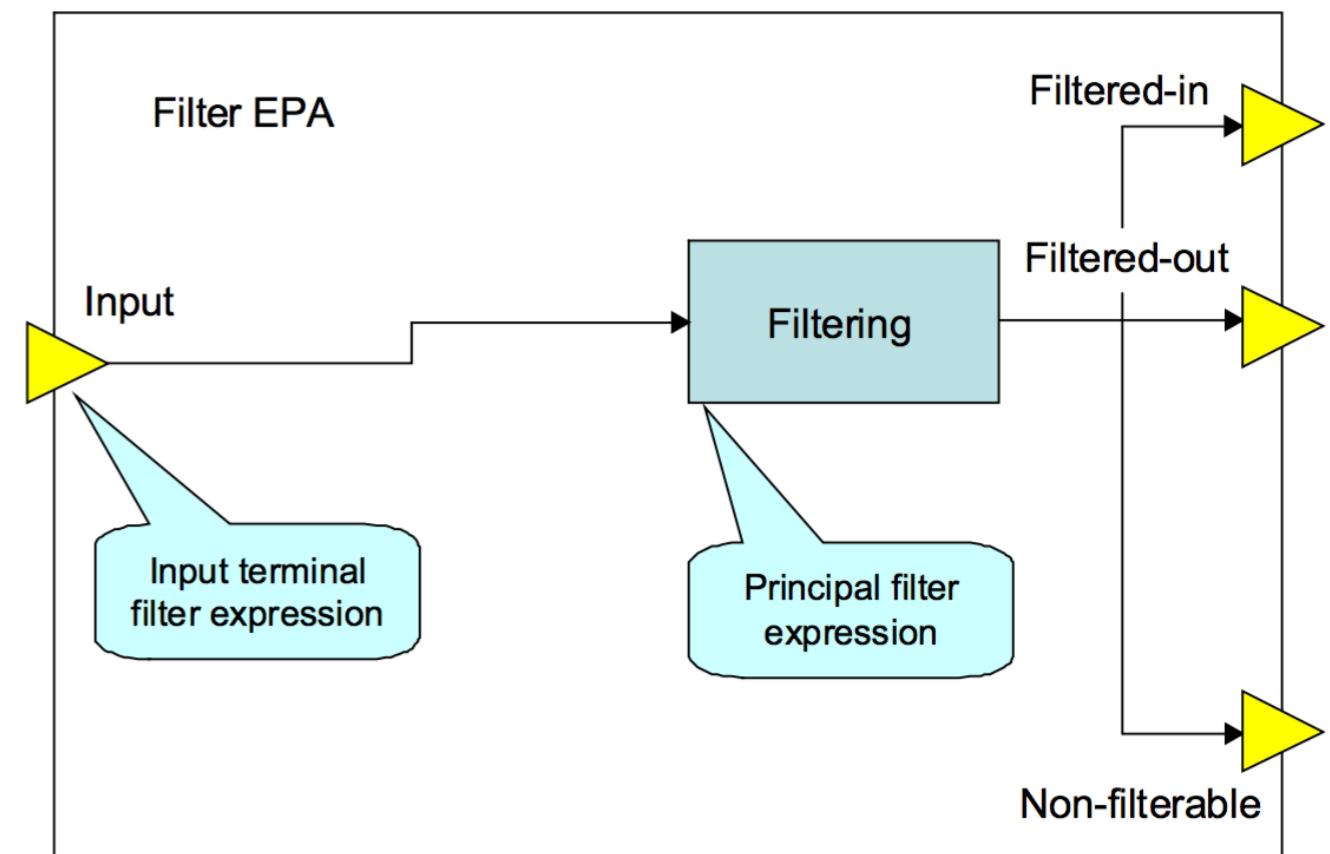
which events are of interest for

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: $\langle e_1, e_2, \dots, e_n \rangle$

- ◆ 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, \langle dismissed patient, admitted patient \rangle is detected whenever a patient is admitted less than 2 days after being dismissed

Time dimension patterns

- **Sequence: $\langle e_1, e_2, \dots, e_n \rangle$**
 - ◆ 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, \langle dismissed patient, admitted patient \rangle 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 - Siddhi Architecture

- **Siddhi is a software library that can be utilized:**

- ◆ Run as a server on its own
- ◆ Run within WSO2 SP as a service
- ◆ Embedded into any Java or Python based application
- ◆ Run on an Android application

WSO2 - Siddhi Architecture

■ **Siddhi is a software library that can be utilized:**

- ◆ Run as a server on its own
- ◆ Run within WSO2 SP as a service
- ◆ Embedded into any Java or Python based application
- ◆ Run on an Android application

■ **Some design decisions**

- ◆ **Event by event processing** of real-time streaming data to achieve low latency.

WSO2 - Siddhi Architecture

■ Siddhi is a software library that can be utilized:

- ◆ Run as a server on its own
- ◆ Run within WSO2 SP as a service
- ◆ Embedded into any Java or Python based application
- ◆ Run on an Android application

■ Some design decisions

- ◆ **Event by event processing** of real-time streaming data to achieve low latency.
- ◆ Achieve high performance by **processing all events in-memory** and by storing their states in-memory.

WSO2 - Siddhi Architecture

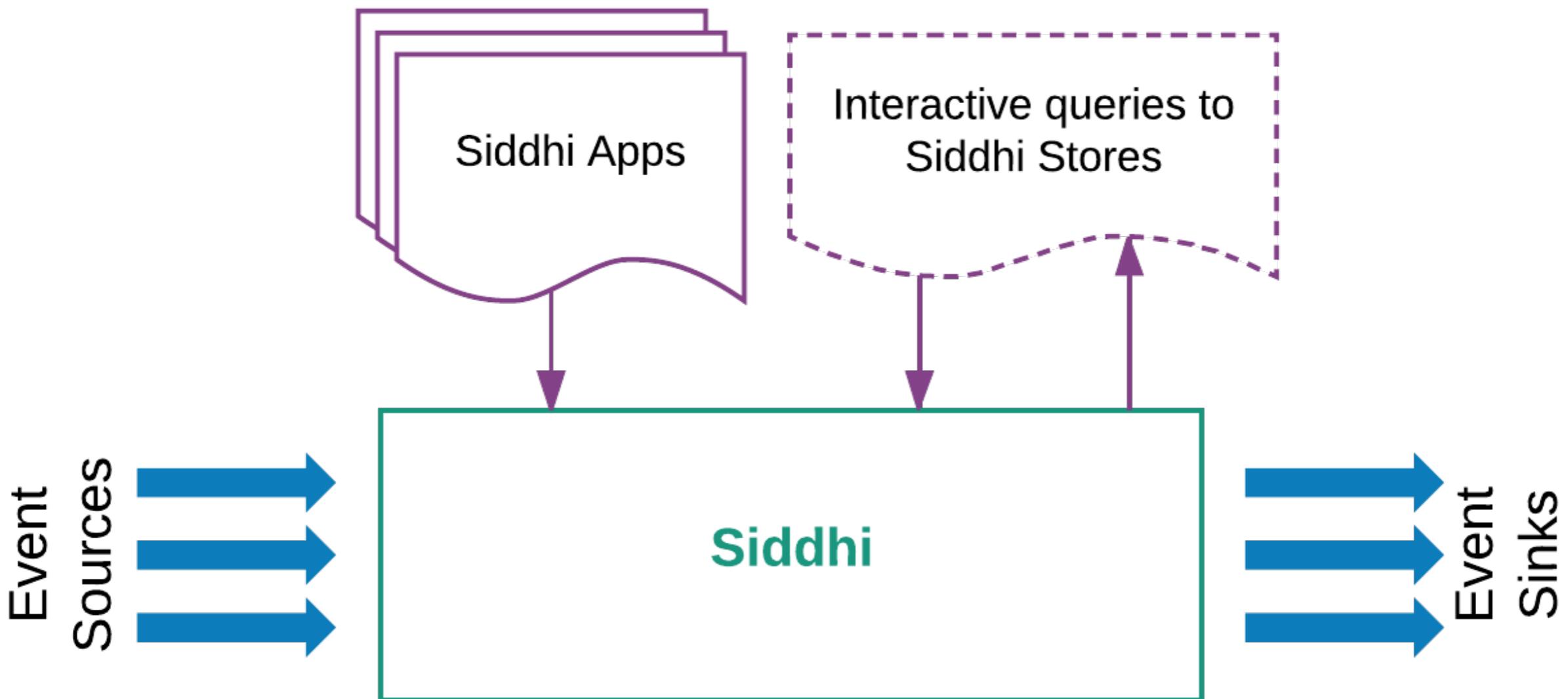
■ Siddhi is a software library that can be utilized:

- ◆ Run as a server on its own
- ◆ Run within WSO2 SP as a service
- ◆ Embedded into any Java or Python based application
- ◆ Run on an Android application

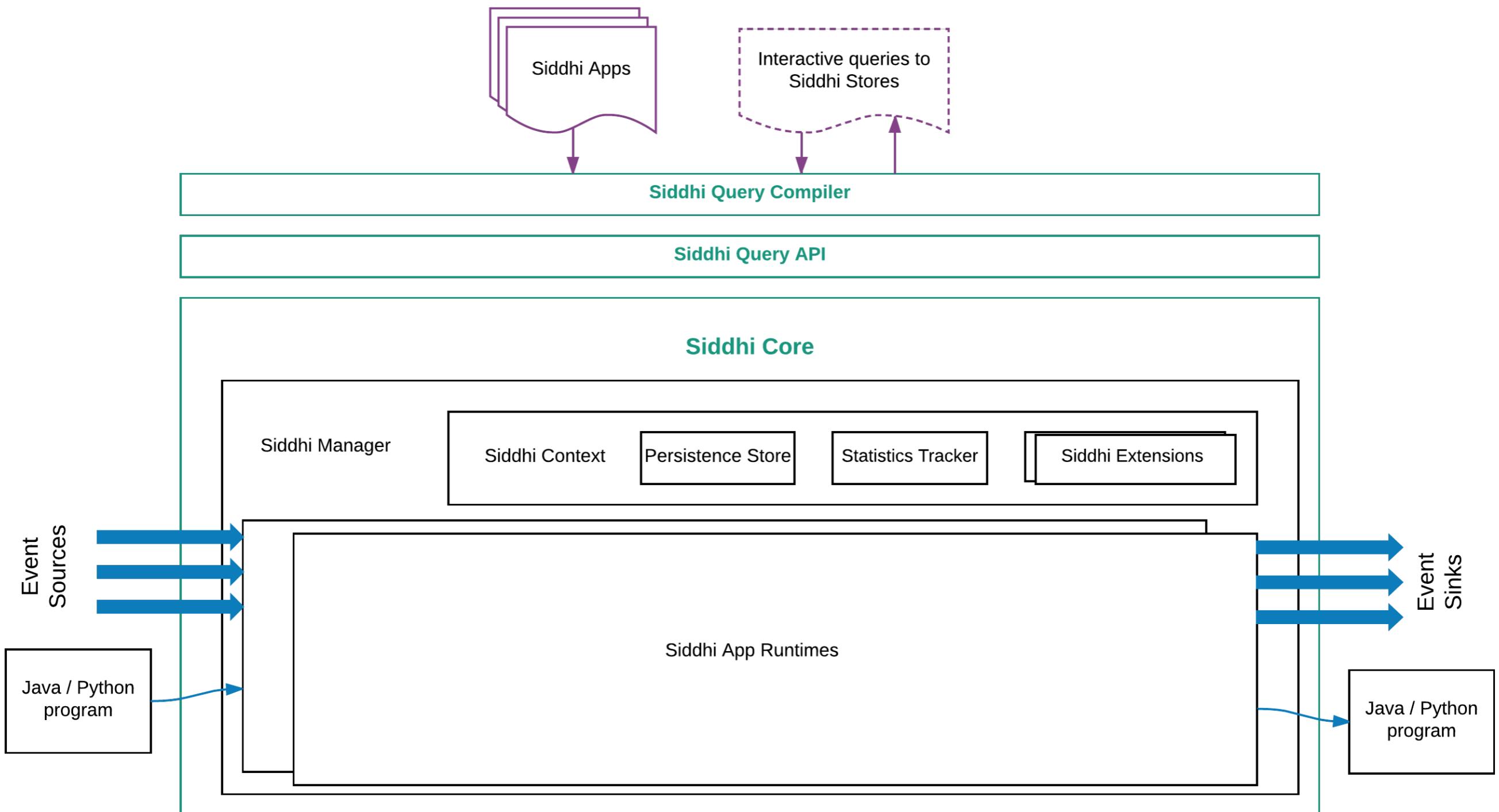
■ Some design decisions

- ◆ **Event by event processing** of real-time streaming data to achieve low latency.
- ◆ Achieve high performance by **processing all events in-memory** and by storing their states in-memory.
- ◆ **Supporting multiple extension points** to accommodate a diverse set of functionality such as supporting multiple sources, sinks, functions, aggregation operations, windows, etc.

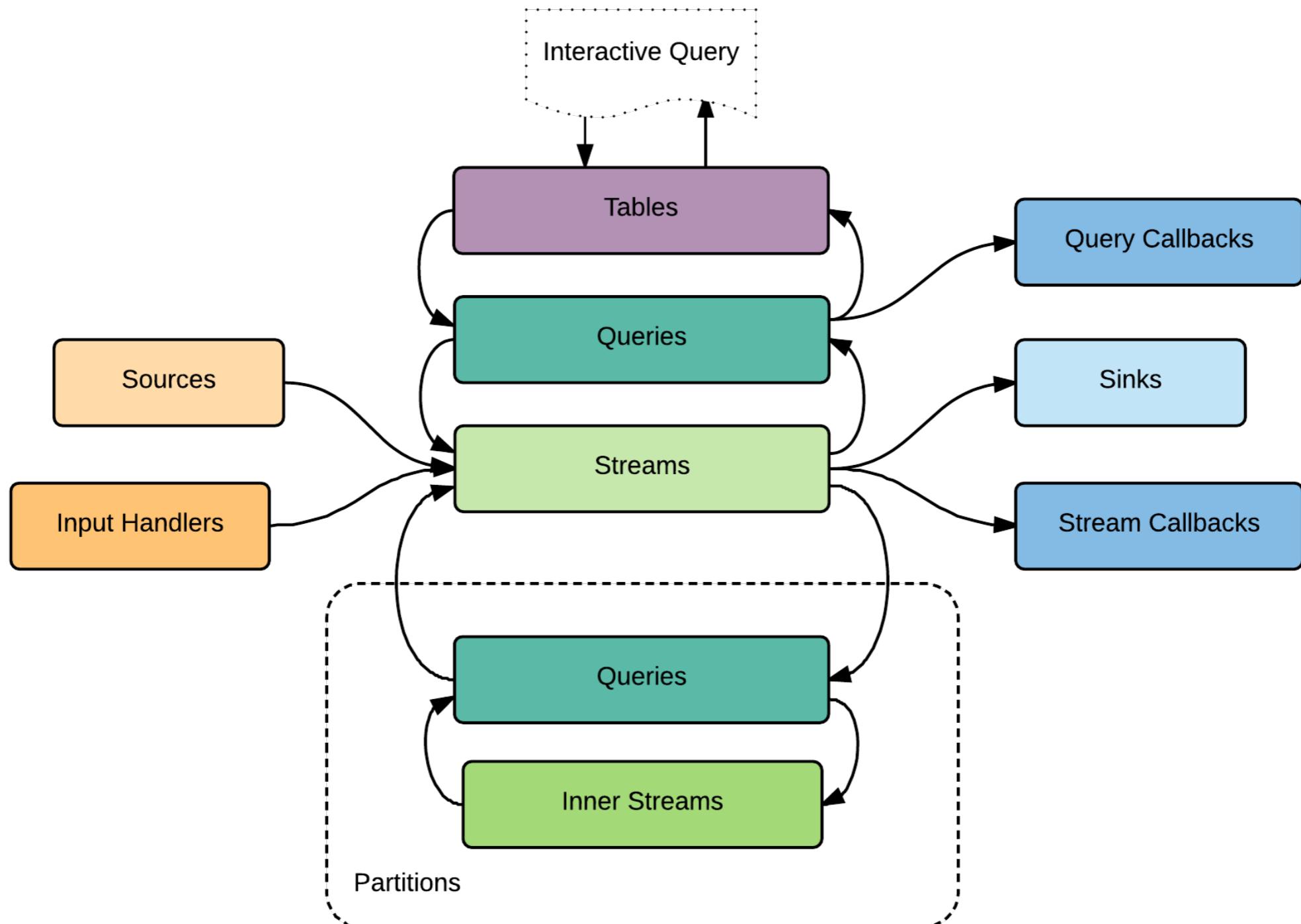
WSO2 - High Level Architecture



WSO2 - Siddhi Component Architecture

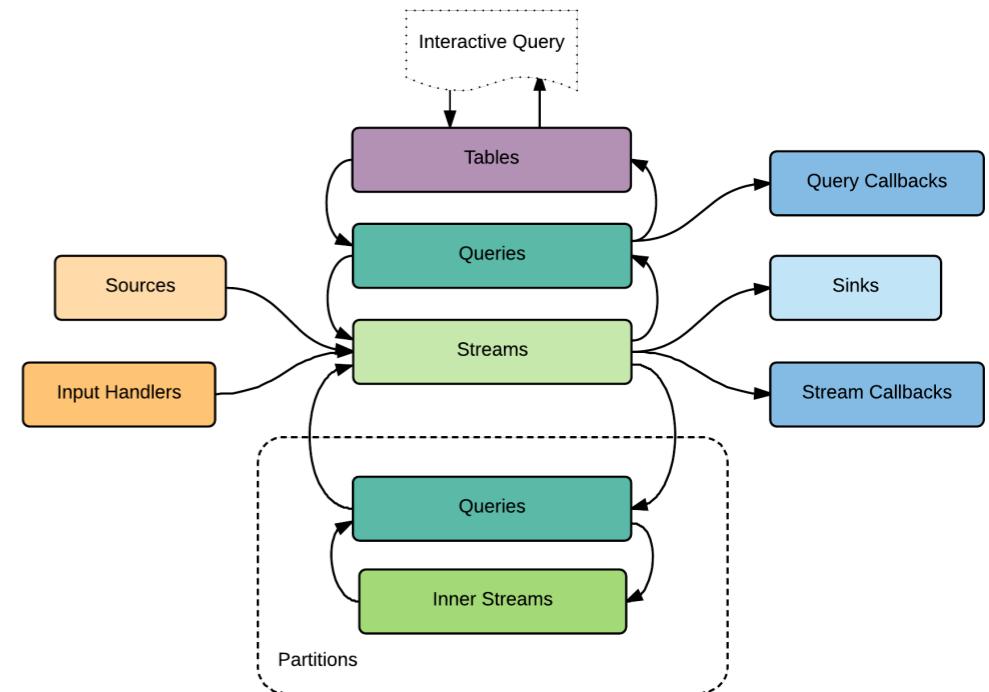


WSO2 - Siddhi Architecture



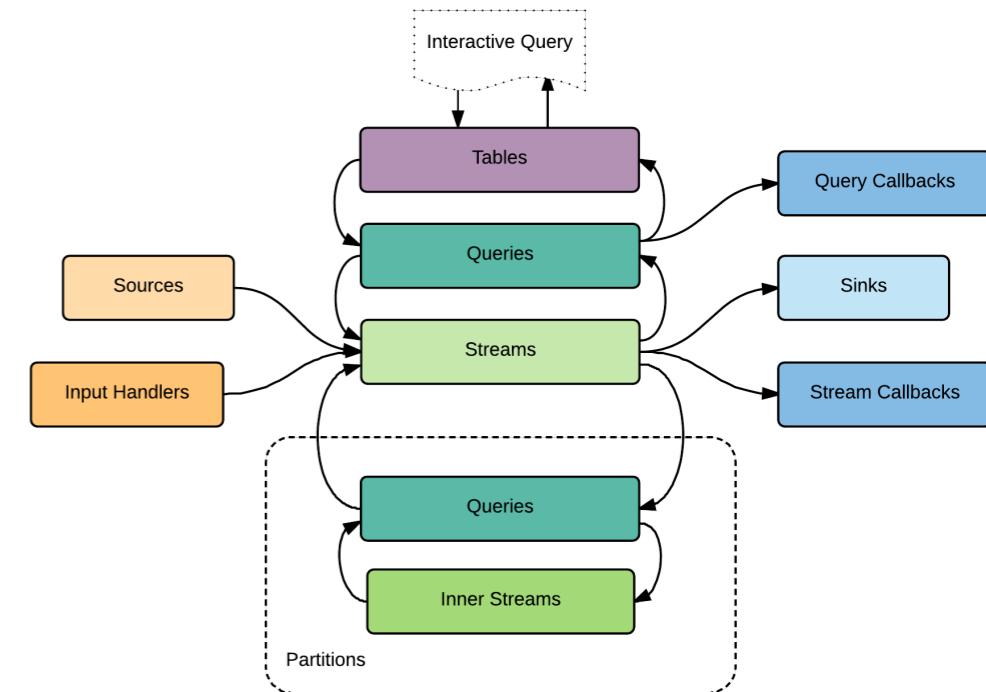
WSO2 - Siddhi Architecture

- **Stream** - A logical series of events ordered in time with a uniquely identifiable name, and set of defined attributes with specific data types defining its schema.



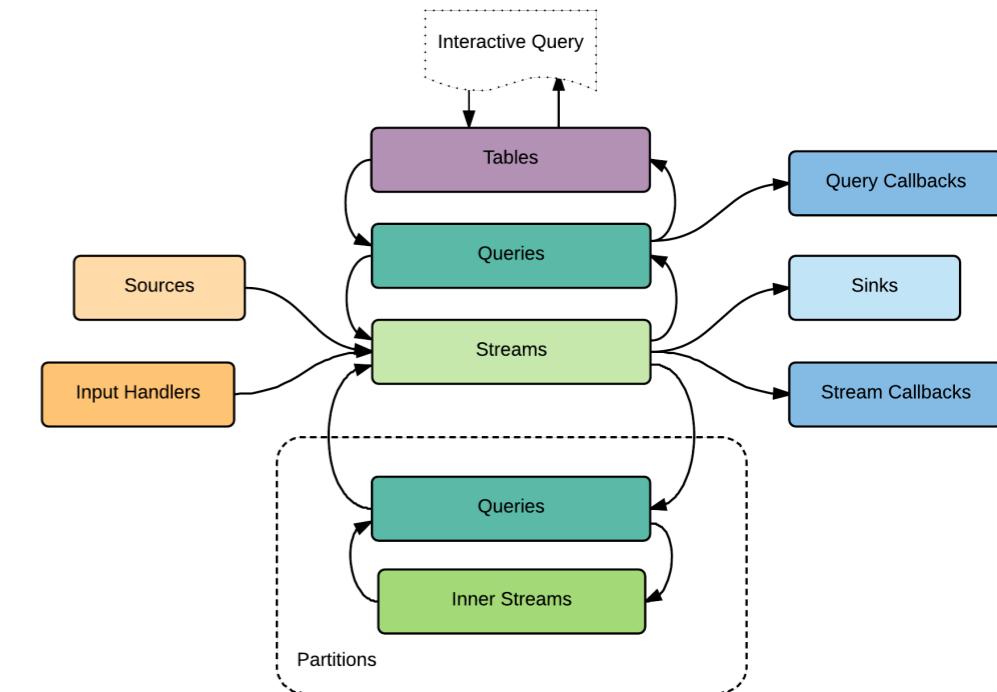
WSO2 - Siddhi Architecture

- **Stream** - A logical series of events ordered in time with a uniquely identifiable name, and set of defined attributes with specific data types defining its schema.
- **Event** - An event is associated with only one stream, and all events of that stream have an identical set of attributes that are assigned specific types (or the same schema). An event contains a timestamp and set of attribute values according to the schema.



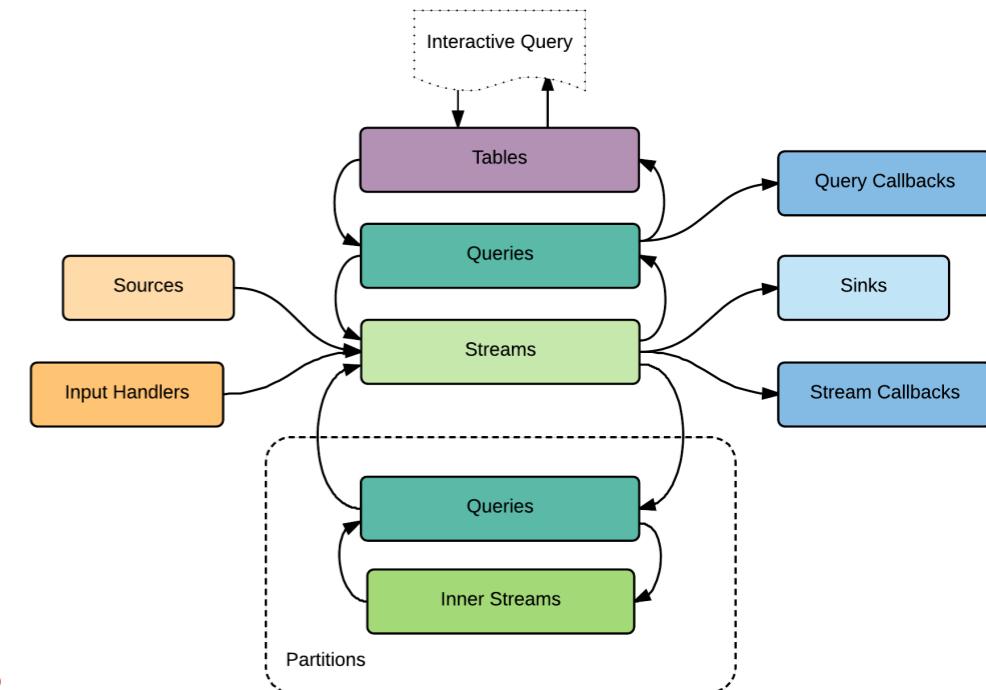
WSO2 - Siddhi Architecture

- **Stream** - A logical series of events ordered in time with a uniquely identifiable name, and set of defined attributes with specific data types defining its schema.
- **Event** - An event is associated with only one stream, and all events of that stream have an identical set of attributes that are assigned specific types (or the same schema). An event contains a timestamp and set of attribute values according to the schema.
- **Table** - A structured representation of data stored with a defined schema. Stored data can be backed by In-Memory, RDBMs, MongoDB, etc. to be accessed and manipulated at runtime.



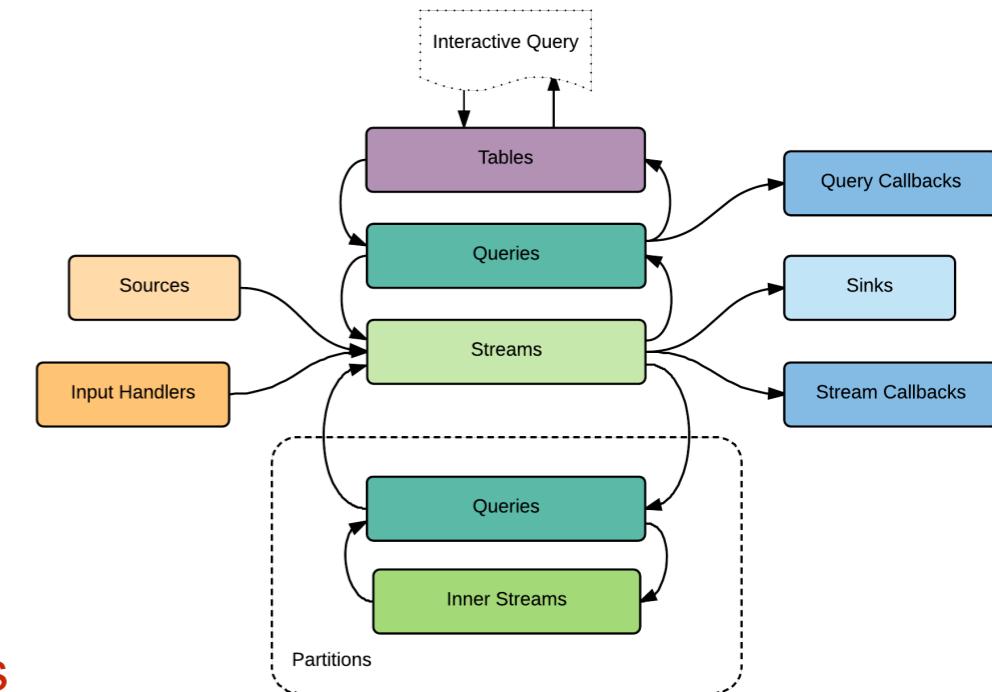
WSO2 - Siddhi Architecture

- **Query** - A logical construct that processes events in streaming manner by combining existing streams and/or tables, and generates events to an output stream or table. A query consumes **one or more input streams**, and **zero or one table**. Then it processes these events in a streaming manner and publishes the output events to streams or tables for further processing or to generate notifications.



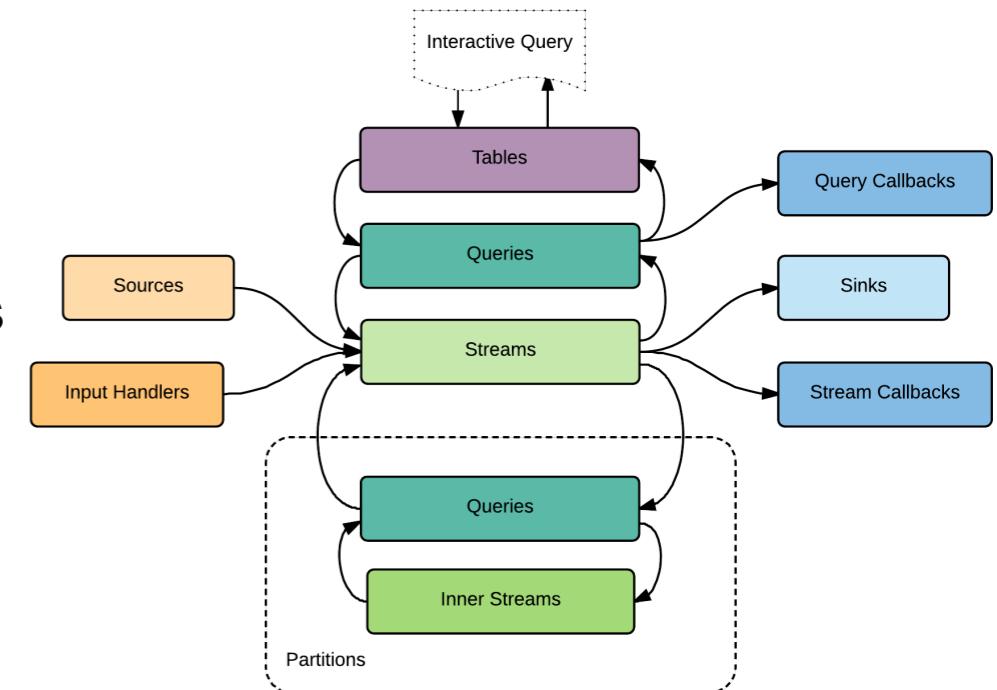
WSO2 - Siddhi Architecture

- **Query** - A logical construct that processes events in streaming manner by combining existing streams and/or tables, and generates events to an output stream or table. A query consumes **one or more input streams**, and **zero or one table**. Then it processes these events in a streaming manner and publishes the output events to streams or tables for further processing or to generate notifications.
- **Source** - A contract that consumes data from external sources (such as TCP, Kafka, HTTP, etc) in the form of events, then converts each event (which can be in XML, JSON, binary, etc. format) to a **Siddhi event**, and passes that to a Stream for processing.



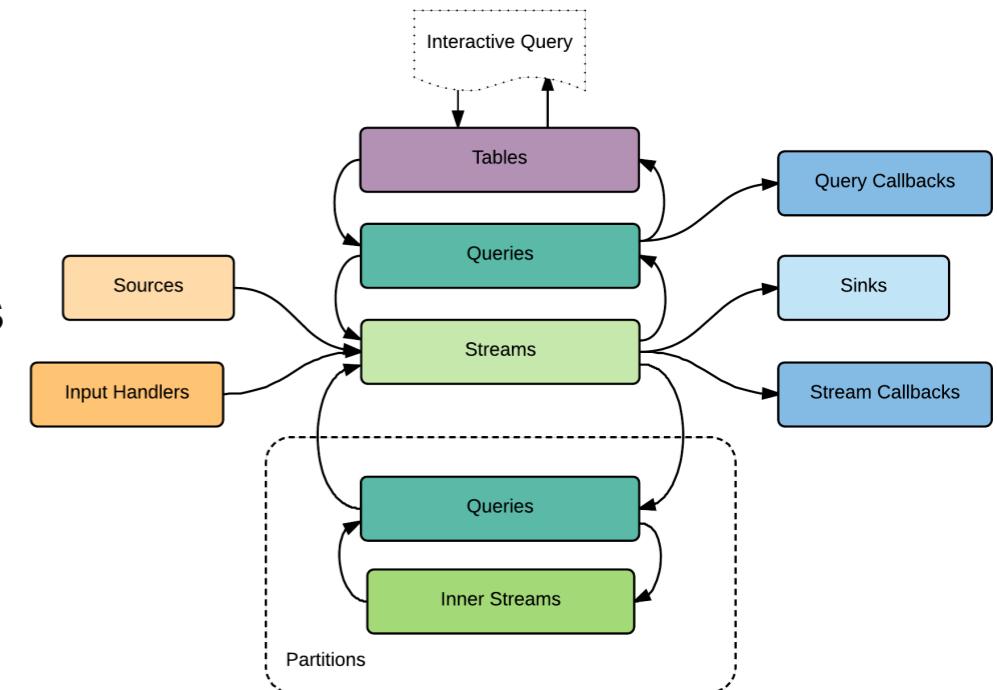
WSO2 - Siddhi Architecture

- **Shink** - A contract that takes events arriving at a stream, maps them to a predefined data format (such as XML, JSON, binary, etc), and publishes them to external endpoints (such as E-mail, TCP, Kafka, HTTP, etc).



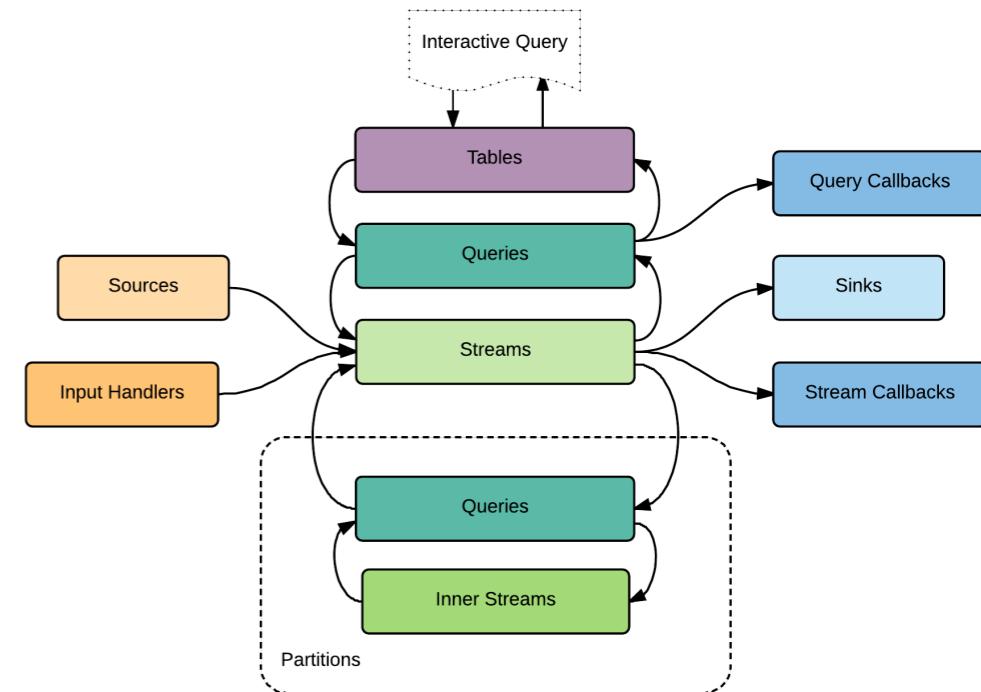
WSO2 - Siddhi Architecture

- **Shink** - A contract that takes events arriving at a stream, maps them to a predefined data format (such as XML, JSON, binary, etc), and publishes them to external endpoints (such as E-mail, TCP, Kafka, HTTP, etc).
- **Input Handler** - A mechanism to programmatically inject events into streams.



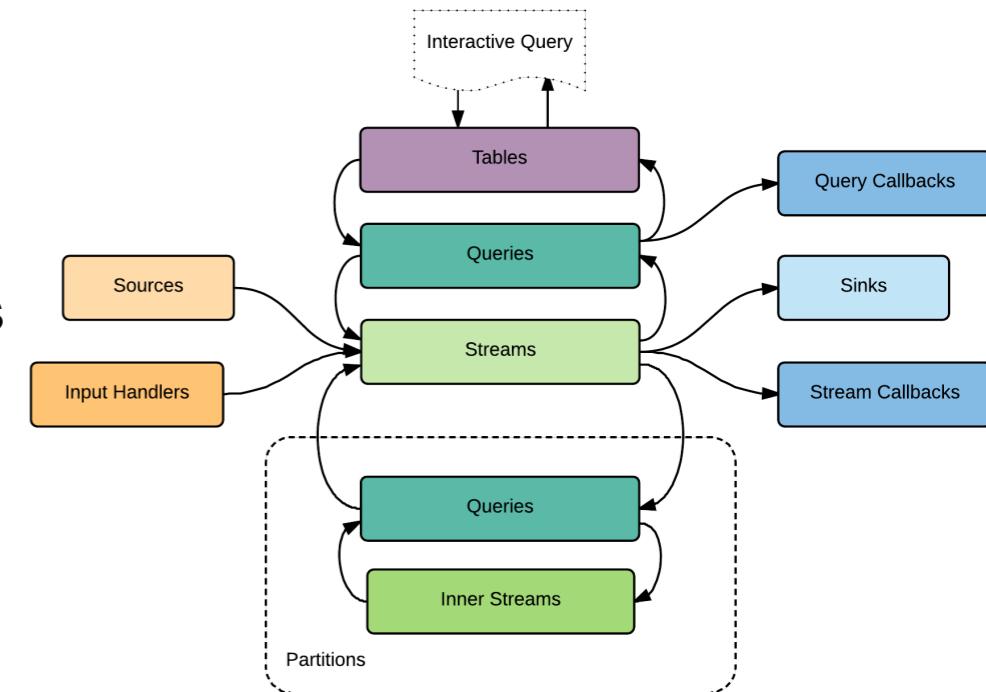
WSO2 - Siddhi Architecture

- **Shink** - A contract that takes events arriving at a stream, maps them to a predefined data format (such as XML, JSON, binary, etc), and publishes them to external endpoints (such as E-mail, TCP, Kafka, HTTP, etc).
- **Input Handler** - A mechanism to programmatically inject events into streams.
- **Stream/Query Callback** - A mechanism to programmatically consume output events from streams and queries.



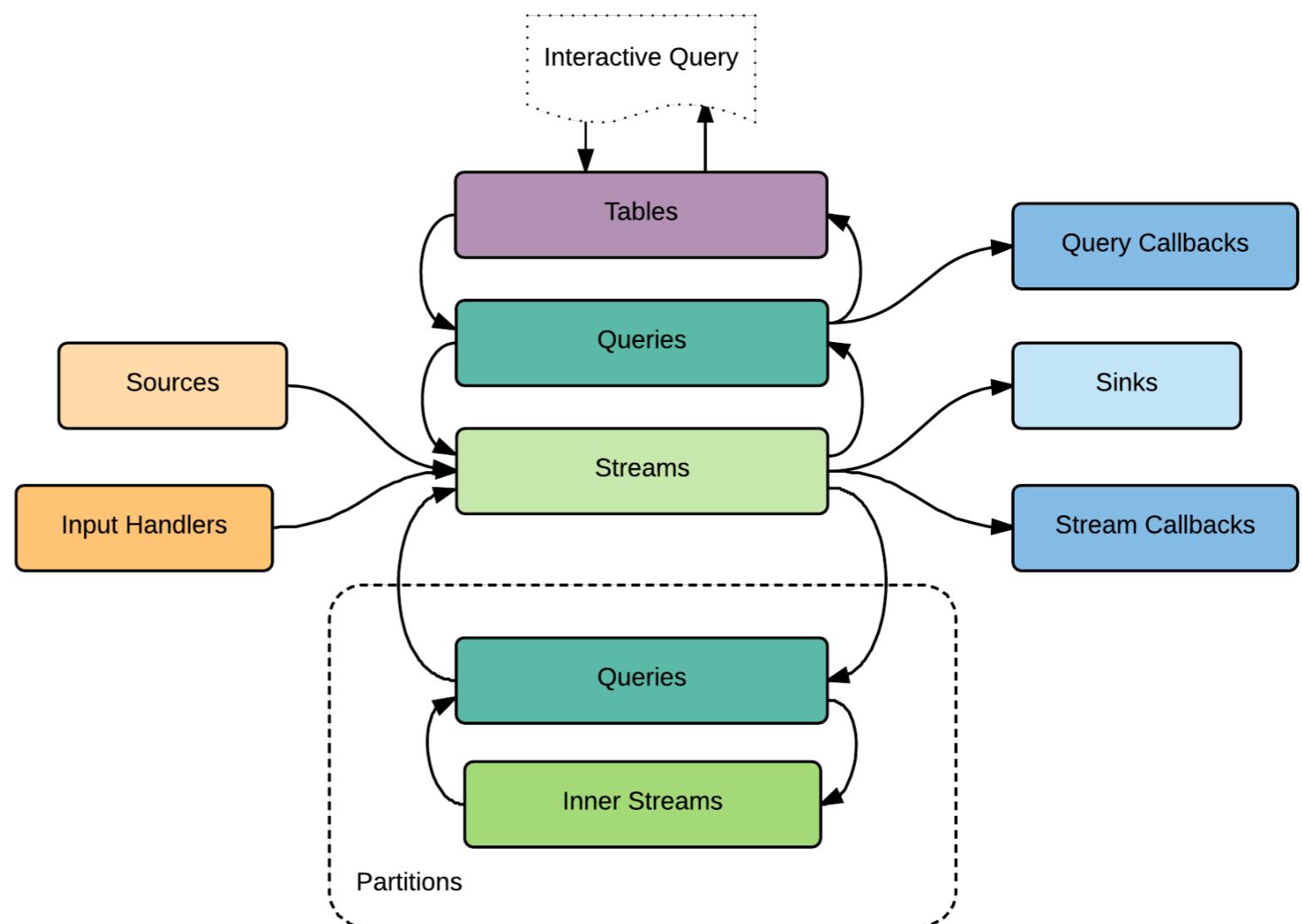
WSO2 - Siddhi Architecture

- **Shink** - A contract that takes events arriving at a stream, maps them to a predefined data format (such as XML, JSON, binary, etc), and publishes them to external endpoints (such as E-mail, TCP, Kafka, HTTP, etc).
- **Input Handler** - A mechanism to programmatically inject events into streams.
- **Stream/Query Callback** - A mechanism to programmatically consume output events from streams and queries.
- **Partition** - A logical container that isolates the processing of queries based on partition keys. Here, a separate instance of queries is generated for each partition key to achieve isolation.



WSO2 - Siddhi Architecture

- **Partition** - A logical container that **isolates the processing of queries based on partition keys**.
Here, a separate instance of queries is generated for each partition key to achieve isolation.
- **Inner Stream** - A positionable stream that connects portioned queries within their partitions, preserving isolation



- **Siddhi QL is the language used by WSO₂ 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)

- **Siddhi QL is the language used by WSO₂ 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)

- **Siddhi QL is the language used by WSO₂ 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 WSO₂, 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 WSO₂, 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)

This can be STRING, INT, LONG, DOUBLE, FLOAT, BOOL or OBJECT.

Event stream

- An event stream is an unbound sequence of event objects
- All events in a stream have the same type
 - ◆ Elsewhere in WSO₂, 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)

This can be STRING, INT, LONG, DOUBLE, FLOAT, BOOL or OBJECT.

- E.g.

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

Source

- Source allows Siddhi to consume events from external systems, and map the events to adhere to the associated stream.

```
@source(type='source_type', static.option.key1='static_option_value1',
        static.option.keyN='static_option_valueN',
        @map(type='map_type', static.option_key1='static_option_value1',
              static.option.keyN='static_option_valueN',
              @attributes( attributeN='attribute_mapping_N',
                           attribute1='attribute_mapping_1')
        )
)
define stream StreamName (attribute1 Type1, attributeN TypeN);
```

Source

- Source allows Siddhi to consume events from external systems, and map the events to adhere to the associated stream.

```
@source(type='source_type', static.option.key1='static_option_value1',
        static.option.keyN='static_option_valueN',
        @map(type='map_type', static.option_key1='static_option_value1',
              static.option.keyN='static_option_valueN',
              @attributes( attributeN='attribute_mapping_N',
                           attribute1='attribute_mapping_1')
        )
)
define stream StreamName (attribute1 Type1, attributeN TypeN);
```

- Source types that are currently supported:

- ◆ WSO2Event, In-memory,
- ◆ WebSocket, HTTP, TCP, Email, JMS, File,
- ◆ Kafka, RabbitMQ, MQTT, Twitter, Amazon SQS, CDC, Prometheus

Source

- Source allows Siddhi to consume events from external systems, and map the events to adhere to the associated stream.

```
@source(type='source_type', static.option.key1='static_option_value1',
        static.option.keyN='static_option_valueN',
        @map(type='map_type', static.option_key1='static_option_value1',
              static.option.keyN='static_option_valueN',
              @attributes( attributeN='attribute_mapping_N',
                           attribute1='attribute_mapping_1')
        )
)
define stream StreamName (attribute1 Type1, attributeN TypeN);
```

- Supported Mapping Types:

- ◆ XML, TEXT, JSON, Binary, Key Value, CSV, Avro (data serialization system - Apache)

```
@source(type='http', receiver.url='http://0.0.0.0:8080/foo', basic.auth.enabled='true',
        @map(type='json'))
define stream InputStream (name string, age int, country string);
```

- Sinks publish events from the streams via multiple transports to external endpoints in various data formats.

```
@sink(type='sink_type', static_option_key1='static_option_value1',
       dynamic_option_key1='{{dynamic_option_value1}}',
       @map(type='map_type', static_option_key1='static_option_value1',
            dynamic_option_key1='{{dynamic_option_value1}}',
            @payload('payload_mapping'))
      )
)
define stream StreamName (attribute1 Type1, attributeN TypeN)
```

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;
```

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

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

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;
```

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

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

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

Query Projection

Action	Description
Selecting required objects for projection	Selecting only some of the attributes from the input stream to be inserted into an output stream. <pre>from TempStream select roomNo, temp insert into RoomTempStream;</pre>

Query Projection

Action	Description
Selecting required objects for projection	Selecting only some of the attributes from the input stream to be inserted into an output stream. <pre>from TempStream select roomNo, temp insert into RoomTempStream;</pre>
Selecting all attributes for projection	Selecting all the attributes in an input stream to be inserted into an output stream. <pre>select * insert into NewTempStream; or from TempStream insert into NewTempStream;</pre>

Query Projection

Action	Description
Selecting required objects for projection	Selecting only some of the attributes from the input stream to be inserted into an output stream. <pre>from TempStream select roomNo, temp insert into RoomTempStream;</pre>
Selecting all attributes for projection	Selecting all the attributes in an input stream to be inserted into an output stream. <pre>select * insert into NewTempStream;</pre> or <pre>from TempStream insert into NewTempStream;</pre>
Renaming attributes	This selects attributes from the input and inserts them into the output stream with different names. <pre>from TempStream select roomNo as roomNumber, temp as temperature insert into RoomTempStream;</pre>

Query Projection

Action	Description
Selecting required objects for projection	Selecting only some of the attributes from the input stream to be inserted into an output stream. <pre>from TempStream select roomNo, temp insert into RoomTempStream;</pre>
Selecting all attributes for projection	Selecting all the attributes in an input stream to be inserted into an output stream. <pre>select * insert into NewTempStream;</pre> or <pre>from TempStream insert into NewTempStream;</pre>
Renaming attributes	This selects attributes from the input and inserts them into the output stream with different names. <pre>from TempStream select roomNo as roomNumber, temp as temperature insert into RoomTempStream;</pre>
Introducing the constant value	Adds constant values by assigning it to an attribute using `as`. <pre>select roomNo, temp, 'C' as scale insert into RoomTempStream;</pre>

Query Projection

Action	Description
Selecting required objects for projection	Selecting only some of the attributes from the input stream to be inserted into an output stream. <pre>from TempStream select roomNo, temp insert into RoomTempStream;</pre>
Selecting all attributes for projection	Selecting all the attributes in an input stream to be inserted into an output stream. <pre>select * insert into NewTempStream;</pre> or <pre>from TempStream insert into NewTempStream;</pre>
Renaming attributes	This selects attributes from the input and inserts them into the output stream with different names. <pre>from TempStream select roomNo as roomNumber, temp as temperature insert into RoomTempStream;</pre>
Introducing the constant value	Adds constant values by assigning it to an attribute using `as`. <pre>select roomNo, temp, 'C' as scale insert into RoomTempStream;</pre>

Using mathematical and logical expressions

Some Functions

- eventTimestamp
- log
- UUID
- default
- cast
- convert
- ifThenElse
- minimum
- maximum
- coalesce
- instanceOfBoolean
- instanceOfDouble
- instanceOfFloat
- instanceOfInteger
- instanceOfLong
- instanceOfString

Some Functions

- `eventTimestamp` —————> Returns the timestamp of the processed event
- `log`
- `UUID`
- `default`
- `cast`
- `convert`
- `ifThenElse`
- `minimum`
- `maximum`
- `coalesce`
- `instanceOfBoolean`
- `instanceOfDouble`
- `instanceOfFloat`
- `instanceOfInteger`
- `instanceOfLong`
- `instanceOfString`

Some Functions

- `eventTimestamp` → Returns the timestamp of the processed event
- `log`
- `UUID` → Generates a UUID (Universally Unique Identifier).
- `default`
- `cast`
- `convert`
- `ifThenElse`
- `minimum`
- `maximum`
- `coalesce`
- `instanceOfBoolean`
- `instanceOfDouble`
- `instanceOfFloat`
- `instanceOfInteger`
- `instanceOfLong`
- `instanceOfString`

Some Functions

- `eventTimestamp` → Returns the timestamp of the processed event
- `log`
- `UUID` → Generates a UUID (Universally Unique Identifier).
- `default` → Checks if the 'attribute' parameter is null and if so returns the value of the 'default' parameter
- `cast`
- `convert`
- `ifThenElse`
- `minimum`
- `maximum`
- `coalesce`
- `instanceOfBoolean`
- `instanceOfDouble`
- `instanceOfFloat`
- `instanceOfInteger`
- `instanceOfLong`
- `instanceOfString`

```
from TempStream  
select default(temp, 0.0) as temp, roomNum  
insert into StandardTempStream;
```

Some Functions

- `eventTimestamp` → Returns the timestamp of the processed event
- `log`
- `UUID` → Generates a UUID (Universally Unique Identifier).
- `default` → Checks if the 'attribute' parameter is null and if so returns the value of the 'default' parameter
- `cast`
- `convert`
- `ifThenElse`
- `minimum`
- `maximum`
- `coalesce`
- `instanceOfBoolean`
- `instanceOfDouble`
- `instanceOfFloat`
- `instanceOfInteger`
- `instanceOfLong`
- `instanceOfString`

```
from TempStream  
select default(temp, 0.0) as temp, roomNum  
insert into StandardTempStream;
```

→ Checks whether the parameter is an instance of

Function Parameters

■ Time parameters

- ◆ Time is a special parameter that can be defined using the integer time value followed by its unit as <int> <unit>

SampleFunction(1 hour 25 min)

Unit	Syntax
Year	year years
Month	month months
Week	week weeks
Day	day days
Hour	hour hours
Minutes	minute minutes min
Seconds	second seconds sec
Milliseconds	millisecond milliseconds

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 ...
```

- ◆ 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 ...
```

- ◆ 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

Windows (from the manual): time

time

Syntax	<code><event> time(<int long time> windowTime)</code>
Extension Type	Window
Description	A sliding time window that holds events that arrived during the last <code>windowTime</code> period at a given time, and gets updated for each event arrival and expiry.
Parameter	<ul style="list-style-type: none">• <code>windowTime</code>: The sliding time period for which the window should hold events.
Return Type	Returns current and expired events.
Examples	<ul style="list-style-type: none">• <code>time(20)</code> for processing events that arrived within the last 20 milliseconds.• <code>time(2 min)</code> for processing events that arrived within the last 2 minutes.

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

timeBatch

Syntax	<code><event> timeBatch(<int long time> windowTime, <int> startTime)</code>
Extension Type	Window
Description	A batch (tumbling) time window that holds events that arrive during <code>windowTime</code> periods, and gets updated for each <code>windowTime</code> .
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	<ul style="list-style-type: none">• <code>timeBatch(20)</code> processes events that arrive every 20 milliseconds.• <code>timeBatch(2 min)</code> processes events that arrive every 2 minutes.• <code>timeBatch(10 min, 0)</code> 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.• <code>timeBatch(10 min, 1000*60*5)</code> 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	<code><event> length(<int> windowLength)</code>
Extension Type	Window
Description	A sliding length window that holds the last <code>windowLength</code> events at a given time, and gets updated for each arrival and expiry.
Parameter	<ul style="list-style-type: none">• <code>windowLength</code>: The number of events that should be included in a sliding length window.
Return Type	Returns current and expired events.
Examples	<ul style="list-style-type: none">• <code>length(10)</code> for processing the last 10 events.• <code>length(200)</code> for processing the last 200 events.

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

lengthBatch

Syntax	<code><event> lengthBatch(<int> windowLength)</code>
Extension Type	Window
Description	A batch (tumbling) length window that holds a number of events specified as the <code>windowLength</code> . The window is updated each time a batch of events that equals the number specified as the <code>windowLength</code> arrives.
Parameter	<code>windowLength</code> : The number of events the window should tumble.
Return Type	Returns current and expired events.
Examples	<ul style="list-style-type: none">• <code>lengthBatch(10)</code> for processing 10 events as a batch.• <code>lengthBatch(200)</code> for processing 200 events as a batch.

Windows (from the manual): `externalTime`

`externalTime`

Syntax	<code><event> externalTime(<long> timestamp, <int long time> windowTime)</code>
Extension Type	Window
Description	A sliding time window based on external time. It holds events that arrived during the last <code>windowTime</code> period from the external timestamp, and gets updated on every monotonically increasing timestamp.
Parameter	<ul style="list-style-type: none">• <code>windowTime</code>: The sliding time period for which the window should hold events.
Return Type	Returns current and expired events.
Examples	<ul style="list-style-type: none">• <code>externalTime(eventTime, 20)</code> for processing events arrived within the last 20 milliseconds from the <code>eventTime</code>• <code>externalTime(eventTimestamp, 2 min)</code> for processing events arrived within the last 2 minutes from the <code>eventTimestamp</code>

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch
- batch
- timeLength
- length
- lengthBatch
- sort
- frequent
- lossyFrequent
- session
- cron
- externalTime
- externalTimeBatch
- delay

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch
- batch
- timeLength
- length
- lengthBatch
- sort
- frequent
- lossyFrequent
- session
- cron
- externalTime
- externalTimeBatch
- delay

→ Starting at a specified timestamp

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch

- batch

- timeLength

- length

- lengthBatch

- sort

- frequent

- lossyFrequent

- session

- cron

- externalTime

- externalTimeBatch

- delay

A sliding time window that, at a given time holds the last window.length events

→ Starting at a specified timestamp

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch

- batch

- timeLength

- length

- lengthBatch

- sort

- frequent

- lossyFrequent

- session

- cron

- externalTime

- externalTimeBatch

- delay

A sliding time window that, at a given time holds the last window.length events

→ Starting at a specified timestamp

→ A delay window holds events for a specific time period that is regarded as a delay period before processing them.

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch
- batch
- timeLength → A sliding time window that, at a given time holds the last window.length events
- length
- lengthBatch
- sort
- frequent
- lossyFrequent
- session
- cron → This window returns events **processed periodically** as the output in time-repeating patterns, **triggered based on time passing**.
- externalTime → Starting at a specified timestamp
- externalTimeBatch
- delay → A delay window holds events for a specific time period that is regarded as a delay period before processing them.

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch
- batch
- timeLength → A sliding time window that, at a given time holds the last window.length events
- length
- lengthBatch
- sort → This window holds a batch of events that equal the number specified as the windowLength and sorts them in the given order.
- frequent
- lossyFrequent
- session
- cron → This window returns events **processed periodically** as the output in time-repeating patterns, **triggered based on time passing**.
- externalTime → Starting at a specified timestamp
- externalTimeBatch
- delay → A delay window holds events for a specific time period that is regarded as a delay period before processing them.

Windows (from the manual)

Following are some inbuilt windows shipped with Siddhi. For more window types, see execution extensions.

- time
- timeBatch
- batch
- timeLength → A sliding time window that, at a given time holds the last window.length events
- length
- lengthBatch
- sort → This window holds a batch of events that equal the number specified as the windowLength and sorts them in the given order.
- frequent → Returns the latest events with the most frequently occurred value for a given attribute(s).
- lossyFrequent
- session
- cron → This window returns events **processed periodically** as the output in time-repeating patterns, **triggered based on time passing**.
- externalTime → Starting at a specified timestamp
- externalTimeBatch
- delay → A delay window holds events for a specific time period that is regarded as a delay period before processing them.

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](#)

Aggregation over windows - Order By

- **Order By allows you to order the aggregated result in ascending and/or descending order based on specified attributes:**

- ◆ Calculates the average temperature per roomNo and deviceID combination for every 10 minutes, and generate output events by ordering them in the ascending order of the room's avgTemp and then by the descending order of roomNo.

```
from TempStream#window.timeBatch(10 min)
select avg(temp) as avgTemp, roomNo, deviceID
group by roomNo, deviceID
order by avgTemp, roomNo desc
insert into AvgTempStream;
```

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**

Joining streams

■ Supported join types

- ◆ Inner join (join)
- ◆ Left outer join (left outer join)
- ◆ Right outer join (right outer join)
- ◆ Full outer join (full outer join)

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 ($^{\circ}$ F)

```
define stream RegulatorStream  
  (regulatorID long, roomNo int, isOn bool);  
  
# previously defined TempStream
```

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 ($^{\circ}$ F)

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

# previously defined TempStream

from TempStream[temp > 40]#window.time(1 min) as T
  join RegulatorStream[not isOn]#window.length(1) as R
  on T.roomNo == R.roomNo
```

Join example

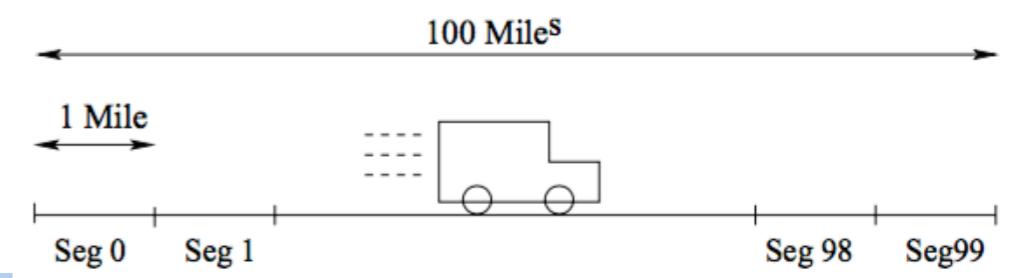
- Emit alerts to switching on temperature regulator, if they are not already on, for a room which has the temperature above 40 degrees ($^{\circ}$ F)

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

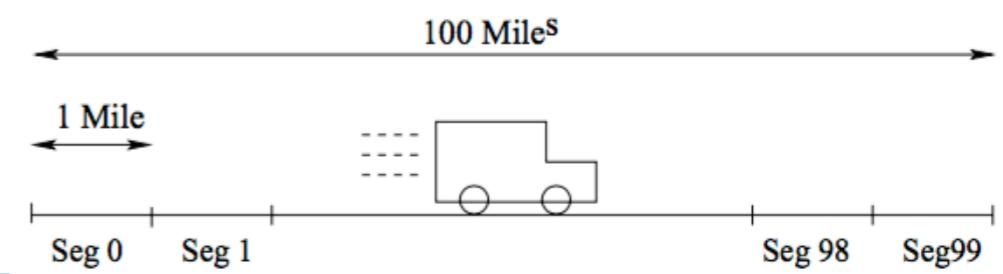
# previously defined TempStream

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

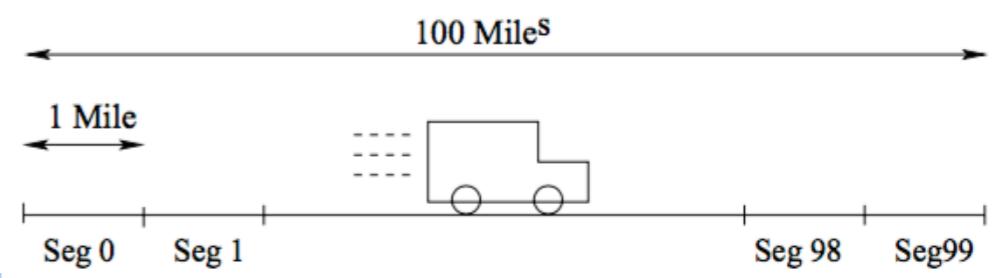


Again the Linear Road



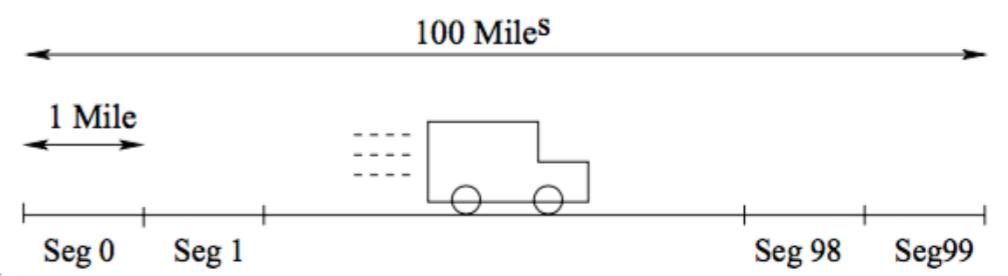
- The road is divided into 100 segments (of 1Km each)

Again the Linear Road



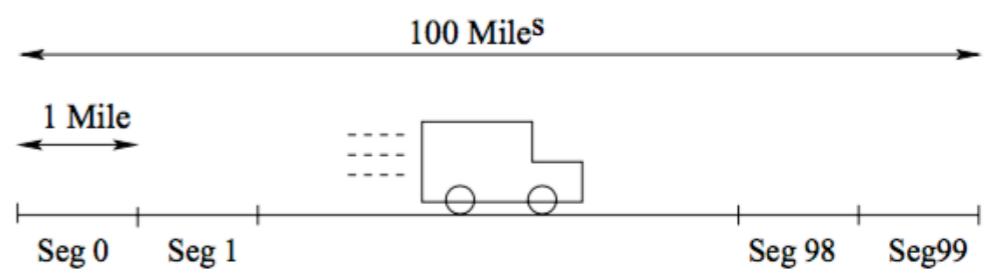
- The road is divided into 100 segments (of 1Km each)
- The input stream has tuples with **carID**, **position**, and **speed**
 - ◆ The position given as the distance in meters from the beginning of the road

Again the Linear Road



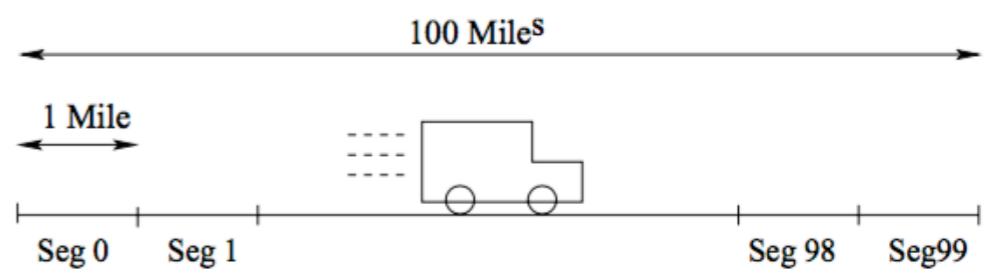
- The road is divided into 100 segments (of 1Km each)
- The input stream has tuples with **carID**, **position**, and **speed**
 - ◆ The position given as the distance in meters from the beginning of the road
- The output stream has tuples with **carID** and **price to pay for a segment**, any time a car enters that segment

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 $2 \times (\text{numCars}-50)^2$, 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 $2 \times (\text{numCars}-50)^2$, 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**

Linear Road in SiddhiQL

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

PosSpeedStr: (carID, position, speed)

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)
```

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)
```

```
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)
```

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**

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)                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)
```

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)

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)           CarSegEntryStr: (carID, segNo)
select carID, segNo
insert into CarSegEntryStr;
```

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

```
from SegSpeedStr#window.time(30 s)
```

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)           CarSegEntryStr: (carID, segNo)
select carID, segNo
insert into CarSegEntryStr;
```

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

```
from SegSpeedStr#window.time(30 s)           CarNumStream: (segNo, count)
```

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 signaling, at any time, the numbers of congested segments

SegSpeedStr: (carID, speed, segNo)

```
from SegSpeedStr#window.time(5 min)
```

Linear Road in SiddhiQL

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

SegSpeedStr: (carID, speed, segNo)

```
from SegSpeedStr#window.time(5 min)
```

CongSegStr: (segNo)

Linear Road in SiddhiQL

- **CongSegStr** as the stream signaling, 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 signaling, 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)

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

```
from CarSegEntryStr
```

CarSegEntryStr: (carID, segNo)

Linear Road in SiddhiQL

- **CongSegStr** as the stream signaling, 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

CarSegEntryStr: (carID, segNo)

TollChargeStr: (carID, Toll)

Linear Road in SiddhiQL

- **CongSegStr** as the stream signaling, 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;
```

CarSegEntryStr: (carID, segNo)

CongSegStr: (segNo)

CarNumStream: (segNo, numCars)

TollChargeStr: (carID, Toll)

Linear Road in SiddhiQL

PosSpeedStr: (carID, position, speed)

Linear Road in SiddhiQL

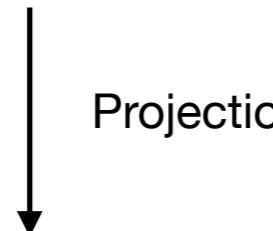
PosSpeedStr: (carID, position, speed)



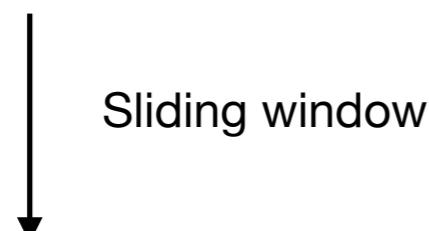
SegSpeedStr: (carID, speed, **segNo**)

Linear Road in SiddhiQL

PosSpeedStr: (carID, position, speed)



SegSpeedStr: (carID, speed, **segNo**)



– cars entering the segments

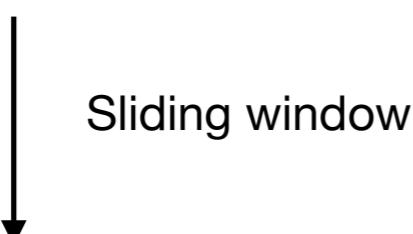
CarSegEntryStr: (carID, segNo)

Linear Road in SiddhiQL

PosSpeedStr: (carID, position, speed)



SegSpeedStr: (carID, speed, **segNo**)



– cars entering the segments
CarSegEntryStr: (carID, segNo)

– number of cars in a segment
CarNumStream: (segNo, count)

Sliding window
Aggregation
group by

Linear Road in SiddhiQL

PosSpeedStr: (carID, position, speed)

Projection

SegSpeedStr: (carID, speed, **segNo**)

Sliding window

– cars entering the segments

CarSegEntryStr: (carID, segNo)

Sliding window
Aggregation
group by

– number of cars in a segment

CarNumStream: (segNo, count)

Sliding window
Aggregation
group by and having

– congested segments

CongSegStr: (segNo)

Linear Road in SiddhiQL

PosSpeedStr: (carID, position, speed)

Projection

SegSpeedStr: (carID, speed, **segNo**)

Sliding window

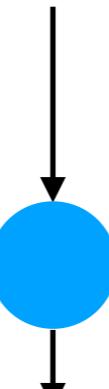
– cars entering the segments
CarSegEntryStr: (carID, segNo)

Sliding window
Aggregation
group by

– number of cars in a segment
CarNumStream: (segNo, count)

Sliding window
Aggregation
group by and having

– congested segments
CongSegStr: (segNo)



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;
```

Time depend patterns

- Up to now, (almost) everything that we've seen in SiddhiQL could be done in a DSMS language, such as CQL (and, arguably, in a more elegant way)
- What really makes the concept of event useful (as opposed to data streams) is the possibility correlate events over time

Time depend patterns

- Up to now, (almost) everything that we've seen in SiddhiQL could be done in a DSMS language, such as CQL (and, arguably, in a more elegant way)
- What really makes the concept of event useful (as opposed to data streams) is **the possibility correlate events over time**
- In Siddhi this is done with so-called *patterns*

Time depend patterns

- Up to now, (almost) everything that we've seen in SiddhiQL could be done in a DSMS language, such as CQL (and, arguably, in a more elegant way)
- What really makes the concept of event useful (as opposed to data streams) is **the possibility correlate events over time**
- In Siddhi this is done with so-called *patterns*
 - ◆ Patterns detect **sequences of events based on arrival order**

Time depend patterns

- Up to now, (almost) everything that we've seen in SiddhiQL could be done in a DSMS language, such as CQL (and, arguably, in a more elegant way)
- What really makes the concept of event useful (as opposed to data streams) is the possibility correlate events over time
- In Siddhi this is done with so-called *patterns*
 - ◆ Patterns detect sequences of events based on arrival order
 - ◆ $E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n$ is a pattern that is detected at the same time of E_n , whenever E_1 was detected, E_2 was detected some time after that, ... and E_n was detected
 - On each of the E_i one can impose conditions dependent on the previous events

Time depend patterns

- Up to now, (almost) everything that we've seen in SiddhiQL could be done in a DSMS language, such as CQL (and, arguably, in a more elegant way)
- What really makes the concept of event useful (as opposed to data streams) is the possibility correlate events over time
- In Siddhi this is done with so-called *patterns*
 - ◆ Patterns detect sequences of events based on arrival order
 - ◆ $E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n$ is a pattern that is detected at the same time of E_n , whenever E_1 was detected, E_2 was detected some time after that, ... and E_n was detected
 - On each of the E_i one can impose conditions dependent on the previous events
 - ◆ For pragmatic reasons it helps to restrict patterns to be detected with a given time gap.

Sequence patterns

- The general form of a pattern is:

```
from every <input ref> = <stream>[<filter>]  
    -> <input ref> = <stream>[<filter>]  
    -> ...  
    within <time gap>  
select ...
```

- ◆ -> correlates **events that arrived with zero or more events in the middle**
- ◆ The `input ref` is just used for reference in the subsequent filters

Sequence patterns

■ The general form of a pattern is:

```
from every <input ref> = <stream>[<filter>]  
    -> <input ref> = <stream>[<filter>]  
    -> ...  
    within <time gap>  
select ...
```

- ◆ -> correlates **events that arrived with zero or more events in the middle**
 - ◆ The `input ref` is just used for reference in the subsequent filters
- ## ■ Alert if, within 10 minutes, the temperature increases more than 5 degrees

Sequence patterns

■ The general form of a pattern is:

```
from every <input ref> = <stream>[<filter>]  
    -> <input ref> = <stream>[<filter>]  
    -> ...  
    within <time gap>  
select ...
```

- ◆ -> correlates **events that arrived with zero or more events in the middle**
 - ◆ The `input ref` is just used for reference in the subsequent filters
- ## ■ Alert if, within 10 minutes, the temperature increases more than 5 degrees

```
from every e1 = TempStream
```

Sequence patterns

■ The general form of a pattern is:

```
from every <input ref> = <stream>[<filter>]
    -> <input ref> = <stream>[<filter>]
    -> ...
    within <time gap>
select ...
```

- ◆ -> correlates **events that arrived with zero or more events in the middle**
 - ◆ The `input ref` is just used for reference in the subsequent filters
- ## ■ Alert if, within 10 minutes, the temperature increases more than 5 degrees

```
from every e1 = TempStream
    -> e2 = TempStream[e1.roomNo == e2.roomNo and e2.temp >
(e1.temp + 5)]
```

Sequence patterns

■ The general form of a pattern is:

```
from every <input ref> = <stream>[<filter>]
    -> <input ref> = <stream>[<filter>]
    -> ...
    within <time gap>
select ...
```

- ◆ -> correlates **events that arrived with zero or more events in the middle**
 - ◆ The `input ref` is just used for reference in the subsequent filters
- ## ■ Alert if, within 10 minutes, the temperature increases more than 5 degrees

```
from every e1 = TempStream
    -> e2 = TempStream[e1.roomNo == e2.roomNo and e2.temp >
        (e1.temp + 5)]
    within 10 min
```

Sequence patterns

■ The general form of a pattern is:

```
from every <input ref> = <stream>[<filter>]
    -> <input ref> = <stream>[<filter>]
    -> ...
    within <time gap>
select ...
```

- ◆ -> correlates **events that arrived with zero or more events in the middle**
 - ◆ The `input ref` is just used for reference in the subsequent filters
- ## ■ Alert if, within 10 minutes, the temperature increases more than 5 degrees

```
from every e1 = TempStream
    -> e2 = TempStream[e1.roomNo == e2.roomNo and e2.temp >
        (e1.temp + 5)]
    within 10 min
select e1.roomNo, e1.temp as IniTemp, e2.temp as FTemp
insert into AlertIncreasingStream
```

Logical patterns

- The general form of conjunction and disjunction pattern (seen before) can also be used in SiddhiQL
 - ◆ This is done by having patterns such as $E_1 \rightarrow (E_2 \text{ and } E_3) \rightarrow \dots \rightarrow E_n$ is

Logical patterns

- The general form of conjunction and disjunction pattern (seen before) can also be used in SiddhiQL
 - ◆ This is done by having patterns such as $E_1 \rightarrow (E_2 \text{ and } E_3) \rightarrow \dots \rightarrow E_n$ is
- In $E_1 \rightarrow (E_2 \text{ and } E_3)$ a pattern is detected if **E_1 is detected and after that both E_2 and E_3 are detected** (without any particular order between them). The time of detection of the complex event is the time of the **latest** of E_2 or E_3

Logical patterns

- The general form of conjunction and disjunction pattern (seen before) can also be used in SiddhiQL
 - ◆ This is done by having patterns such as $E_1 \rightarrow (E_2 \text{ and } E_3) \rightarrow \dots \rightarrow E_n$ is
- In $E_1 \rightarrow (E_2 \text{ and } E_3)$ a pattern is detected if **E_1 is detected and after that both E_2 and E_3 are detected** (without any particular order between them). The time of detection of the complex event is the time of the **latest** of E_2 or E_3
- In $E_1 \rightarrow (E_2 \text{ or } E_3)$ a pattern is detected if **E_1 is detected and after that either E_2 or E_3 is detected**. The time of detection of the complex event is the time of the **earliest** of E_2 or E_3

Logical patterns

- The general form of conjunction and disjunction pattern (seen before) can also be used in SiddhiQL
 - ◆ This is done by having patterns such as $E_1 \rightarrow (E_2 \text{ and } E_3) \rightarrow \dots \rightarrow E_n$ is
- In $E_1 \rightarrow (E_2 \text{ and } E_3)$ a pattern is detected if **E_1 is detected and after that both E_2 and E_3 are detected** (without any particular order between them). The time of detection of the complex event is the time of the **latest** of E_2 or E_3
- In $E_1 \rightarrow (E_2 \text{ or } E_3)$ a pattern is detected if **E_1 is detected and after that either E_2 or E_3 is detected**. The time of detection of the complex event is the time of the **earliest** of E_2 or E_3
- The E_1 can also be absent. I.e, you can have a pattern such as $(E_2 \text{ or } E_3)$, or with *and*, meaning that an event is detected whenever either E_2 or E_3 occur

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

```
define stream RegulatorStream  
(regulatorID long, roomNo int, temp double);
```

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

```
define stream RegulatorStream  
(regulatorID long, roomNo int, temp double);  
  
from every e1 = RegulatorStream -> e2 = TempStream
```

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

```
define stream RegulatorStream
(regulatorID long, roomNo int, temp double);

from every e1 = RegulatorStream -> e2 = TempStream
-> e3 = TempStream[e1.roomNo == e3.roomNo and e3.temp >= e1.temp]
```

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

```
define stream RegulatorStream
(regulatorID long, roomNo int, temp double);

from every e1 = RegulatorStream -> e2 = TempStream
-> e3 = TempStream[e1.roomNo == e3.roomNo and e3.temp >= e1.temp]
or e4 = TempStream[e2.roomNo==e4.roomNo and e4.temp>(e2.temp+5)]
```

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

```
define stream RegulatorStream
(regulatorID long, roomNo int, temp double);

from every e1 = RegulatorStream -> e2 = TempStream
-> e3 = TempStream[e1.roomNo == e3.roomNo and e3.temp >= e1.temp]
or e4 = TempStream[e2.roomNo==e4.roomNo and e4.temp>(e2.temp+5)]
within 10 min
```

Logical pattern example

- Alert when the **room temperature reaches the temperature set on the regulator, or increases by more than 5 degrees**, over the last 10 minutes

TempStream: (deviceID, roomNo, temp)

```
define stream RegulatorStream
(regulatorID long, roomNo int, temp double);

from every e1 = RegulatorStream -> e2 = TempStream
-> e3 = TempStream[e1.roomNo == e3.roomNo and e3.temp >= e1.temp]
or e4 = TempStream[e2.roomNo==e4.roomNo and e4.temp>(e2.temp+5)]
within 10 min
select e1.roomNo
insert into AlertRoomTemp
```

More sequence patterns

- Besides the sequences with $E1 \rightarrow E2$, which **allows zero or more events occurring between E1 and E2**, there are also sequences **(E1, E2)** which are detected whenever **E1 is immediately followed by E2**

More sequence patterns

- Besides the sequences with $E1 \rightarrow E2$, which **allows zero or more events occurring between E1 and E2**, there are also sequences **(E1, E2)** which are detected whenever **E1 is immediately followed by E2**
- It is also possible to use **usual operators from regular expressions** (*, + and ?) to correlate the occurrence of events

More sequence patterns

- Besides the sequences with **E1 -> E2**, which **allows zero or more events occurring between E1 and E2**, there are also sequences **(E1, E2)** which are detected whenever **E1 is immediately followed by E2**
- It is also possible to use **usual operators from regular expressions** (*, + and ?) to correlate the occurrence of events
- **For example, identify peak temperatures**

```
from every e1 = TempStream
      , e2 = TempStream[e2[last-1].temp <= temp or e1.temp <= temp] +
      , e3 = TempStream[e3.temp < e2[last].temp]
select e1.temp as InitialTemp, e2.[last] as PeakTemp
insert into TempPeakStream
```

More sequence patterns

- Besides the sequences with **E1 -> E2**, which **allows zero or more events occurring between E1 and E2**, there are also sequences **(E1, E2)** which are detected whenever **E1 is immediately followed by E2**
- It is also possible to use **usual operators from regular expressions** (*, + and ?) to correlate the occurrence of events
- **For example, identify peak temperatures**

```
from every e1 = TempStream
    , e2 = TempStream[e2[last-1].temp <= temp or e1.temp <= temp] +
    , e3 = TempStream[e3.temp < e2[last].temp]
select e1.temp as InitialTemp, e2.[last] as PeakTemp
insert into TempPeakStream
```

More sequence patterns

```
from every e1 = TempStream
    , e2 = TempStream[e2[last-1].temp <= temp or e1.temp <= temp] +
    , e3 = TempStream[e3.temp < e2[last].temp]
select e1.temp as InitialTemp, e2.[last] as PeakTemp
insert into TempPeakStream
```

More sequence patterns

■ For example, identify peak temperatures

```
from every e1 = TempStream
    , e2 = TempStream[e2[last-1].temp <= temp or e1.temp <= temp] +
    , e3 = TempStream[e3.temp < e2[last].temp]
select e1.temp as InitialTemp, e2.[last] as PeakTemp
insert into TempPeakStream
```

- note that the peaks are only detected after the peak temperature

More sequence patterns

■ For example, identify peak temperatures

```
from every e1 = TempStream
    , e2 = TempStream[e2[last-1].temp <= temp or e1.temp <= temp] +
    , e3 = TempStream[e3.temp < e2[last].temp]
select e1.temp as InitialTemp, e2.[last] as PeakTemp
insert into TempPeakStream
```

- note that the peaks are only detected after the peak temperature
- the e2[last] denotes the last of the several possible occurrences of e2 (given the +)

More sequence patterns

■ For example, identify peak temperatures

```
from every e1 = TempStream
    , e2 = TempStream[e2[last-1].temp <= temp or e1.temp <= temp] +
    , e3 = TempStream[e3.temp < e2[last].temp]
select e1.temp as InitialTemp, e2.[last] as PeakTemp
insert into TempPeakStream
```

- note that the peaks are only detected after the peak temperature
- the e2[last] denotes the last of the several possible occurrences of e2 (given the +)
 - ◆ **Besides last, one can use numbers**
 - ◆ E.g. e2[3] denotes the **third occurrence of e2 in the + sequence**

Another Linear road variant

- If a car is detected in a segment N, and the last two times it was detected was in segment N+1, within a time frame of 30 minutes, then warn the police

Another Linear road variant

- If a car is detected in a segment N, and the last two times it was detected was in segment N+1, within a time frame of 30 minutes, then warn the police

SegSpeedStr: (carID, speed, segNo)

```
from every e1 = SegSpeedStr
```

Another Linear road variant

- If a car is detected in a segment N, and the last two times it was detected was in segment N+1, within a time frame of 30 minutes, then warn the police

SegSpeedStr: (carID, speed, segNo)

```
from every e1 = SegSpeedStr  
      , e2 = SegSpeedStr[e1.carID==e2.carID and e1(segNo==e2.segNo)]
```

Another Linear road variant

- If a car is detected in a segment N, and the last two times it was detected was in segment N+1, within a time frame of 30 minutes, then warn the police

SegSpeedStr: (carID, speed, segNo)

```
from every e1 = SegSpeedStr
      , e2 = SegSpeedStr[e1.carID==e2.carID and e1(segNo==e2.segNo]
      , e3 = SegSpeedStr[e2.carID==e3.carID and e2.segNo==e3.segNo+1]
```

Another Linear road variant

- If a car is detected in a segment N, and the last two times it was detected was in segment N+1, within a time frame of 30 minutes, then warn the police

SegSpeedStr: (carID, speed, segNo)

```
from every e1 = SegSpeedStr
    , e2 = SegSpeedStr[e1.carID==e2.carID and e1(segNo==e2.segNo]
    , e3 = SegSpeedStr[e2.carID==e3.carID and e2.segNo==e3.segNo+1]
    within 30 min
select e3.carID, e3.segNo
insert into WarnWrongDirectionStream
```

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 < n:m > \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 < n:m > \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted
- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 <n:m> \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted
- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
```

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 <n:m> \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted
- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
```

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 <n:m> \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted
- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
```

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 <n:m> \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted
- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
```

Counting sequence patterns

- One can also have sequences with a **fixed number of occurrences** of a given event
- $E1 \rightarrow E2 \langle n:m \rangle \rightarrow E3$ happens at the same time of $E3$, when $E1$ occurs before $E3$, and $E2$ occurs in-between at least n times and at most m times
 - ◆ Either one of n and m can be omitted
- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```

Counting sequence patterns

- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```

Counting sequence patterns

- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```

$t_{e1} \dots \dots \dots$

Counting sequence patterns

- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```

t_{e1}

t_{e2}

Counting sequence patterns

- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```

t_{e1}

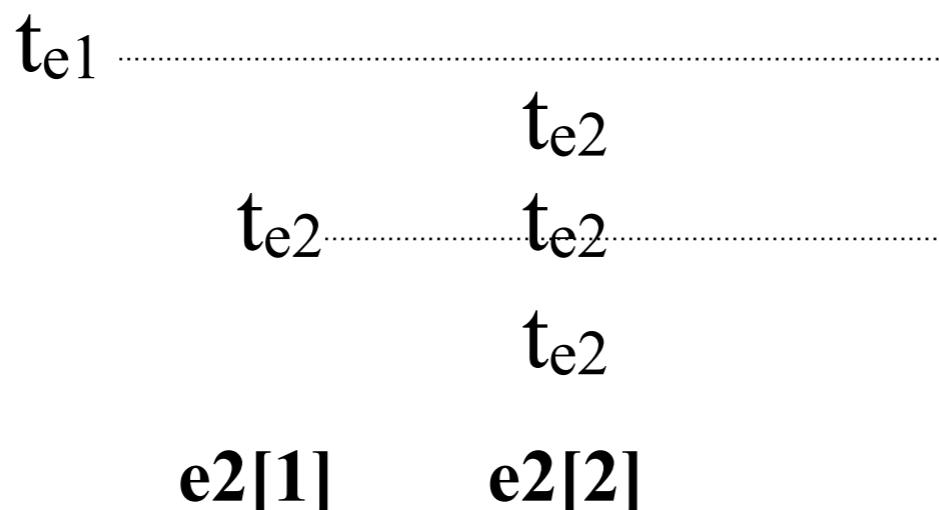
t_{e2}

e2[1]

Counting sequence patterns

- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

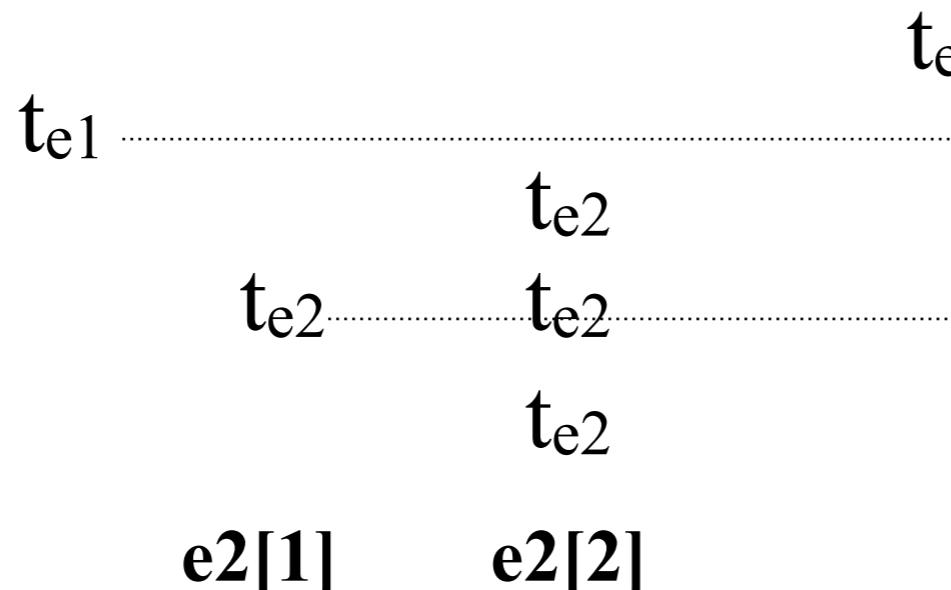
```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```



Counting sequence patterns

- For example, identify whenever there is an increase in the temperature in a room, with exactly two decreasing temperature measures values in-between.

```
from every e1 = TempStream , e2 = TempStream[e2.temp < e1.temp]<2:2>
      , e3 = TempStream[e3.temp > e1.temp]
select e1.temp as InitialTemp, e2[1].temp as FirstMiddle,
      e2[2].temp as SecondMiddle, e3.temp as Final
insert into TempIncrWith2DecreaseStream
```



Partitions

- Sometimes it helps to partition a stream by a set of attributes, or by a range, and process each of the partitioned stream in parallel. SiddhiQL allows this kind of partition:

```
partition with ( <attribute> of <stream>, ... )  
begin  
    <Query>  
    ...  
end;
```

Partitions

- Sometimes it helps to partition a stream by a set of attributes, or by a range, and process each of the partitioned stream in parallel. SiddhiQL allows this kind of partition:

```
partition with ( <attribute> of <stream>, ... )
begin
    <Query>
    ...
end;
```

- Per sensor, compute the maximum temperature over the last 10 temperature events

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

Hotel, Car, Flight example

- Detect when, for a given client, there is both an event of hotel, car and flight reservation (very simplified)

Hotel, Car, Flight example

- Detect when, for a given client, there is both an event of hotel, car and flight reservation (very simplified)

```
partition with ( clientID of ReservationStream )
begin
    from e1 = ReservationStream[reservationType == 'car'] and
          e2 = ReservationStream[reservationType == 'hotel'] and
          e3 = ReservationStream[reservationType == 'flight']
    select clientID
    insert into ReadyClientStream
end;
```

Yet another LR variant

- If a car is detected in segment N and then in segment N+2, and in-between it is never detected in segment N+1, then signal a possible failure

Yet another LR variant

- If a car is detected in segment N and then in segment N+2, and in-between it is never detected in segment N+1, then signal a possible failure

```
partition with (carID of SegSpeedStr)
begin
    from every e1 = SegSpeedStr ,
          e2 = SegSpeedStr[e2(segNo == e1(segNo + 2)]
    select carID, segNo, e1(segNo + 1) as failed
    insert into PossibleFailureStr
end;
```

Range Partitions

- Partition can be made over a range of value in attributes:

```
partition with ( <cond> as <partition key> or ...
                <cond> as <partition key>
                of <stream> or ... )

begin
    <Query>
    ...
end;
```

Range Partitions

- Partition can be made over a range of value in attributes:

```
partition with ( <cond> as <partition key> or ...
                <cond> as <partition key>
                of <stream> or ... )
begin
    <Query>
    ...
end;
```

- Per office area compute the average temperature in batches of 10 minutes

Range Partitions

- Partition can be made over a range of value in attributes:

```
partition with ( <cond> as <partition key> or ...
                <cond> as <partition key>
                of <stream> or ... )
begin
    <Query>
    ...
end;
```

- Per office area compute the average temperature in batches of 10 minutes

```
partition with
    ( roomNo >100 as 'serverRooms' or roomNo =<100 as 'officeRooms' of TempStream)
```

Range Partitions

- Partition can be made over a range of value in attributes:

```
partition with ( <cond> as <partition key> or ...
                <cond> as <partition key>
                of <stream> or ... )
begin
    <Query>
    ...
end;
```

- Per office area compute the average temperature in batches of 10 minutes

```
partition with
    ( roomNo >100 as 'serverRooms' or roomNo =<100 as 'officeRooms' of TempStream)
begin
    from TempStream#window.timeBatch(10 min)
    select partitionKey, avg(temp) as AvgTemp
    insert into AverageAreaTempStream
end;
```

Inner Streams

- Inside a partition one can define several queries with inner streams, whose scope is restricted to the partition statement
 - ◆ The inner streams have a name started by #
- Per sensor, compute the maximum temperature over the last 10 measurements, when the sensor has an average temperature greater than 20 over the last minute:

```
partition with (deviceID of TempStream)
begin
    from TempStream#window.time(1 min)
    select deviceID, temp, avg(temp) as AvgTemp
    insert into #AverageTempStr;

    from #AverageTempStr[AvgTemp > 20]#length(10)
    select deviceID, max(temp) as maxTemp
    insert into DeviceTempStream;
end;
```

Inner Streams

■ Inside a partition one can define several queries with inner streams, whose scope is restricted to the partition statement

- ◆ The inner streams have a name started by #
- ◆ Queries inside a partition block can use inner streams to communicate with each other while preserving partition isolation. The output of a query can be used as an input only by another query within in the same partition.
- ◆ These streams cannot be accessed outside a partition block.

Events and Tables

- **Contrary to DSMS, the primary operators of SiddhiQL are stream2stream**
 - ◆ In DSMS, with s2r and r2s operators, dealing together with streams/events and tables was quite natural
- **But SiddhiQL also has operators for dealing together with events and tables**
 - ◆ E.g. for getting the customers ID from car ID, in the highway example

Events and Tables

- **Contrary to DSMS, the primary operators of SiddhiQL are stream2stream**
 - ◆ In DSMS, with s2r and r2s operators, dealing together with streams/events and tables was quite natural
- **But SiddhiQL also has operators for dealing together with events and tables**
 - ◆ E.g. for getting the customers ID from car ID, in the highway example
- **WSO₂ supports own tables, with**

```
define table <table name>
(<attName> <attType>, ..., <attName> <attType>);
```

Events and Tables

- Contrary to DSMS, the primary operators of SiddhiQL are stream2stream
 - ◆ In DSMS, with s2r and r2s operators, dealing together with streams/events and tables was quite natural
- But SiddhiQL also has operators for dealing together with events and tables
 - ◆ E.g. for getting the customers ID from car ID, in the highway example
- WSO₂ supports own tables, with

```
define table <table name>
(<attName> <attType>, ..., <attName> <attType>);
```

- With SiddhiQL one can insert/delete/update tuples from the table
 - ◆ But this is not so useful!
 - ◆ It would be much nicer to access “proper” databases.

Accessing external tables

- WSO₂ allows one to access tables in “proper” DBMSs via JDBC, with

```
@store(type='rdbms', jdbc.url='<URL>', driver.name='<Driver>',
       username='<User>', password='<Pass>', table.name='<OriginalTable>')
define table <table name>
(<attName> <attType>, ..., <attName> <attType>);
```

Accessing external tables

- WSO₂ allows one to access tables in “proper” DBMSs via JDBC, with

```
@store(type='rdbms', jdbc.url='<URL>', driver.name='<Driver>',
       username='<User>', password='<Pass>', table.name='<OriginalTable>')
define table <table name>
(<attName> <attType>, ..., <attName> <attType>);
```

- For example, to access a table with rooms and their types:

```
@store(type='rdbms', jdbc.url='jdbc:mysql://localhost:3306/mydb',
       driver.name='com.mysql.jdbc.Driver',
       username='root', password='admin', table.name='RoomTypes')

define table RoomsTable (roomNo int, type string);
```

Joining tables and events

- One can join a table with an event stream (with restrictions):

Joining tables and events

- **One can join a table with an event stream (with restrictions):**

- ◆ A table can only be joined with a stream to produce a stream without any windows

Joining tables and events

- **One can join a table with an event stream (with restrictions):**
 - ◆ A table can only be joined with a stream to produce a stream without any windows
 - It is **not possible to join more than one table, nor to more than one stream**

Joining tables and events

■ One can join a table with an event stream (with restrictions):

- ◆ A table can only be joined with a stream to produce a stream without any windows
 - It is **not possible to join more than one table, nor to more than one stream**
- ◆ Anytime a new event arrives in the stream, the table is consulted and an event is output to the result stream

Joining tables and events

- **One can join a table with an event stream (with restrictions):**

- ◆ A table can only be joined with a stream to produce a stream without any windows
 - It is **not possible to join more than one table, nor to more than one stream**
 - ◆ Anytime a new event arrives in the stream, the table is consulted and an event is output to the result stream

- **This is mainly used to enrich the stream with attributes that are in a table**

Joining tables and events

- **One can join a table with an event stream (with restrictions):**
 - ◆ A table can only be joined with a stream to produce a stream without any windows
 - It is **not possible to join more than one table, nor to more than one stream**
 - ◆ Anytime a new event arrives in the stream, the table is consulted and an event is output to the result stream
- **This is mainly used to enrich the stream with attributes that are in a table**
- **E.g. add the room type to the temperature events**

```
from TempStream join RoomsTable on (TempStream.roomNo ==  
RoomsTable.roomNo)  
select deviceID, TempStream.roomNo, RoomsTable.type, temp  
insert into EnrichedTempStream;
```

Yet another LR variant

- Instead of just signaling the carID and the amount of the toll, immediately charge the account associated to the car:

Yet another LR variant

- Instead of just signaling the carID and the amount of the toll, immediately charge the account associated to the car:

```
@store(type='rdbms',
       jdbc.url='jdbc:mysql://localhost:3306/carsDB',
       driver.name='com.mysql.jdbc.Driver',
       username='root', password='admin',
       table.name='NIBofCar' )

define table Nibs (nib double, carID int);

from ToolChargeStr join Nibs on (ToolChargeStr.carID ==
Nibs.carID)
select nib, Tool
insert into ChargingStream;
```

Updating tables

- Tables can be updated with SiddhiQL rules
- For inserting tuples in a table, just use the usual SiddhiQL `insert into`, but with a table name rather than a stream name
 - ◆ Whenever an event arrives in the stream, that passes the filters, a tuple is inserted in the table
 - ◆ Windows can be used, as well as `insert current/expired/all`

Updating tables

- Tables can be updated with SiddhiQL rules
- For inserting tuples in a table, just use the usual SiddhiQL `insert into`, but with a table name rather than a stream name
 - ◆ Whenever an event arrives in the stream, that passes the filters, a tuple is inserted in the table
 - ◆ Windows can be used, as well as `insert current/expired/all`
- For example, storing all the occurrences of temperatures higher than 60 in a **HighTemperatures** tables

```
from TempStream [ temp > 60 ]
select *
insert into HighTempsTable;
```

Updating tables (cont)

- For deleting tuples, instead of `insert into`, use

```
delete <table> for <current|expired|all> events  
on <condition>
```

- ◆ You cannot use `delete` on streams (of course!)
- For example, whenever a temperature event expires in a 30 minutes window, delete the tuple of the corresponding table (silly, but...)

```
from TempStream#window.time(30 min)  
delete RoomsTable for expired events  
on RoomsTable.roomNo == TempStream.roomNo
```

Updating tables (cont)

- For deleting tuples, instead of `insert into`, use

```
delete <table> for <current|expired|all> events  
on <condition>
```

- ◆ You cannot use `delete` on streams (of course!)
- For example, whenever a temperature event expires in a 30 minutes window, delete the tuple of the corresponding table (silly, but...)

```
from TempStream#window.time(30 min)  
delete RoomsTable for expired events  
on RoomsTable.roomNo == TempStream.roomNo
```

- The update command is also available with `update... on ... set ...` as usual in databases

Updating tables (cont)

- For deleting tuples, instead of `insert into`, use

```
delete <table> for <current|expired|all> events  
on <condition>
```

- ◆ You cannot use `delete` on streams (of course!)
- For example, whenever a temperature event expires in a 30 minutes window, delete the tuple of the corresponding table (silly, but...)

```
from TempStream#window.time(30 min)  
delete RoomsTable for expired events  
on RoomsTable.roomNo == TempStream.roomNo
```

- The update command is also available with `update... on ... set ...` as usual in databases
- Update can also be made with `insert overwrite... on ...`

Internally raising (time) events

- **Sometimes it helps to generate events**
 - ◆ At least for debugging, it surely does
- **You can raise timely event using so-called SiddhiQL triggers**
 - ◆ A trigger is defined for a time interval, or a cron expression
 - ◆ It generates an event stream with the name of the trigger, and events with a single attribute `triggered_time` of type `long`

Internally raising (time) events

- **Sometimes it helps to generate events**
 - ◆ At least for debugging, it surely does
- **You can raise timely event using so-called SiddhiQL triggers**
 - ◆ A trigger is defined for a time interval, or a cron expression
 - ◆ It generates an event stream with the name of the trigger, and events with a single attribute `triggered_time` of type `long`

■ For example

```
define trigger HourClock at every 60 min;
```

- ◆ See cron at any Unix manual...

Putting it to work in WSO2

- **To put this to work, you have first to deal with the usual bureaucracy of these systems**
 - ◆ Define input event adapters to have the event streams
 - It supports JMS, SOAP, HTTP, WSO2Event, ...
 - ◆ Define output event adapters
 - It supports JMS, SOAP, HTTO, WSO2Event, Email, SMS, Cassandra, ...
 - ◆ Event formatters may be placed between the output streams and the output adapters
 - They do even more bureaucratic stuff, of format translating (XMLs, envelops, etc)
- **Then there is a browser based tool, where you can**
 - ◆ define all these adapters
 - ◆ register SiddhiQL rules
 - ◆ view events/statistics in dashboards

Other approaches to CEP

Different approaches to CEP

- In SiddhiQL events come in event streams, and the language is SQL based
 - ◆ In fact, in the syntax and the names (but not necessarily concepts) it is not that different from DSMSs
- But there are other ways to view CEPs

Different approaches to CEP

- In SiddhiQL events come in event streams, and the language is SQL based
 - ◆ In fact, in the syntax and the names (but not necessarily concepts) it is not that different from DSMSs
- But there are other ways to view CEPs
- Most of these are rule-based languages, and come in several flavours
 - ◆ Production Rules
 - ◆ Active Rules
 - ◆ Logic Programming Rules
- Some aspects of implementation of CEPs, and of underlying semantics, are best understood in these other approaches

Production Rules

- Rules that react on state changes, of the form
 - ◆ *if Condition do Action*
- Developed in AI for Expert Systems, and more recently to Business Rules
 - if awarded mile in last year > 25000 do award silver status
 - if awarded mile in last year > 100000 do award gold status
 - if flight is less than 500 miles do award 500 miles
 - if flights is in business do award 50% miles as bonus
 - if silver status and flight not in partner do award 30% miles as bonus

Production Rules

- Rules that react on state changes, of the form
 - ◆ *if Condition do Action*
- Developed in AI for Expert Systems, and more recently to Business Rules
 - if awarded mile in last year > 25000 do award silver status
 - if awarded mile in last year > 100000 do award gold status
 - if flight is less than 500 miles do award 500 miles
 - if flights is in business do award 50% miles as bonus
 - if silver status and flight not in partner do award 30% miles as bonus
- Forward chaining (from antecedent to consequent) execution
 - ◆ Once an antecedent is satisfied, the rule must be triggered and its consequent executed; this may cause other antecedents to be satisfied; ...

Production Rules

- Rules that react on state changes, of the form
 - ◆ *if Condition do Action*
- Developed in AI for Expert Systems, and more recently to Business Rules
 - if awarded mile in last year > 25000 do award silver status
 - if awarded mile in last year > 100000 do award gold status
 - if flight is less than 500 miles do award 500 miles
 - if flights is in business do award 50% miles as bonus
 - if silver status and flight not in partner do award 30% miles as bonus
- Forward chaining (from antecedent to consequent) execution
 - ◆ Once an antecedent is satisfied, the rule must be triggered and its consequent executed; this may cause other antecedents to be satisfied; ...
- Many business (rule) systems
 - ◆ IBM/ILOG, TIBCO Business Events, Jess, FICO Blaze Advisor, OPS5, ...

Production Rules and Events

Production Rules and Events

- **Conditions may be the occurrence of an event**

- ◆ As if a fact with the event is (temporarily) added to the knowledge base

Production Rules and Events

- **Conditions may be the occurrence of an event**
 - ◆ As if a fact with the event is (temporarily) added to the knowledge base
- **The action may be raising a (derived) event**

Production Rules and Events

- **Conditions may be the occurrence of an event**
 - ◆ As if a fact with the event is (temporarily) added to the knowledge base
- **The action may be raising a (derived) event**
- **The conditions/events may be joined and filtered according to type pattern matching**
 - ◆ If true, the corresponding actions are triggered

Production Rules Execution

- **The execution of Production Rules is typically based on the Rete algorithm (or variants)**
 - ◆ Defined in the 1982 by Charles Forgy, for expert systems
 - ◆ Builds a network of nodes, where each intermediate node corresponds to a condition pattern in the antecedent of a rule
 - root nodes contains classes
 - leafs contain actions
 - there are other intermediate nodes for conjunctions of conditions (reused for several actions/rules)

Production Rules Execution

- **The execution of Production Rules is typically based on the Rete algorithm (or variants)**
 - ◆ Defined in the 1982 by Charles Forgy, for expert systems
 - ◆ Builds a network of nodes, where each intermediate node corresponds to a condition pattern in the antecedent of a rule
 - root nodes contains classes
 - leafs contain actions
 - there are other intermediate nodes for conjunctions of conditions (reused for several actions/rules)
 - ◆ (new) facts are matched against these patterns (incremental evaluation)

Production Rules Execution

- **The execution of Production Rules is typically based on the Rete algorithm (or variants)**
 - ◆ Defined in the 1982 by Charles Forgy, for expert systems
 - ◆ Builds a network of nodes, where each intermediate node corresponds to a condition pattern in the antecedent of a rule
 - root nodes contains classes
 - leafs contain actions
 - there are other intermediate nodes for conjunctions of conditions (reused for several actions/rules)
 - ◆ (new) facts are matched against these patterns (incremental evaluation)
 - ◆ Nodes internally store information about its satisfaction

Production Rules Execution

- **The execution of Production Rules is typically based on the Rete algorithm (or variants)**

- ◆ Defined in the 1982 by Charles Forgy, for expert systems
- ◆ Builds a network of nodes, where each intermediate node corresponds to a condition pattern in the antecedent of a rule
 - root nodes contains classes
 - leafs contain actions
 - there are other intermediate nodes for conjunctions of conditions (reused for several actions/rules)
- ◆ (new) facts are matched against these patterns (incremental evaluation)
- ◆ Nodes internally store information about its satisfaction
- ◆ Once a node is totally satisfied, the corresponding action(s) is triggered

Production Rules Execution

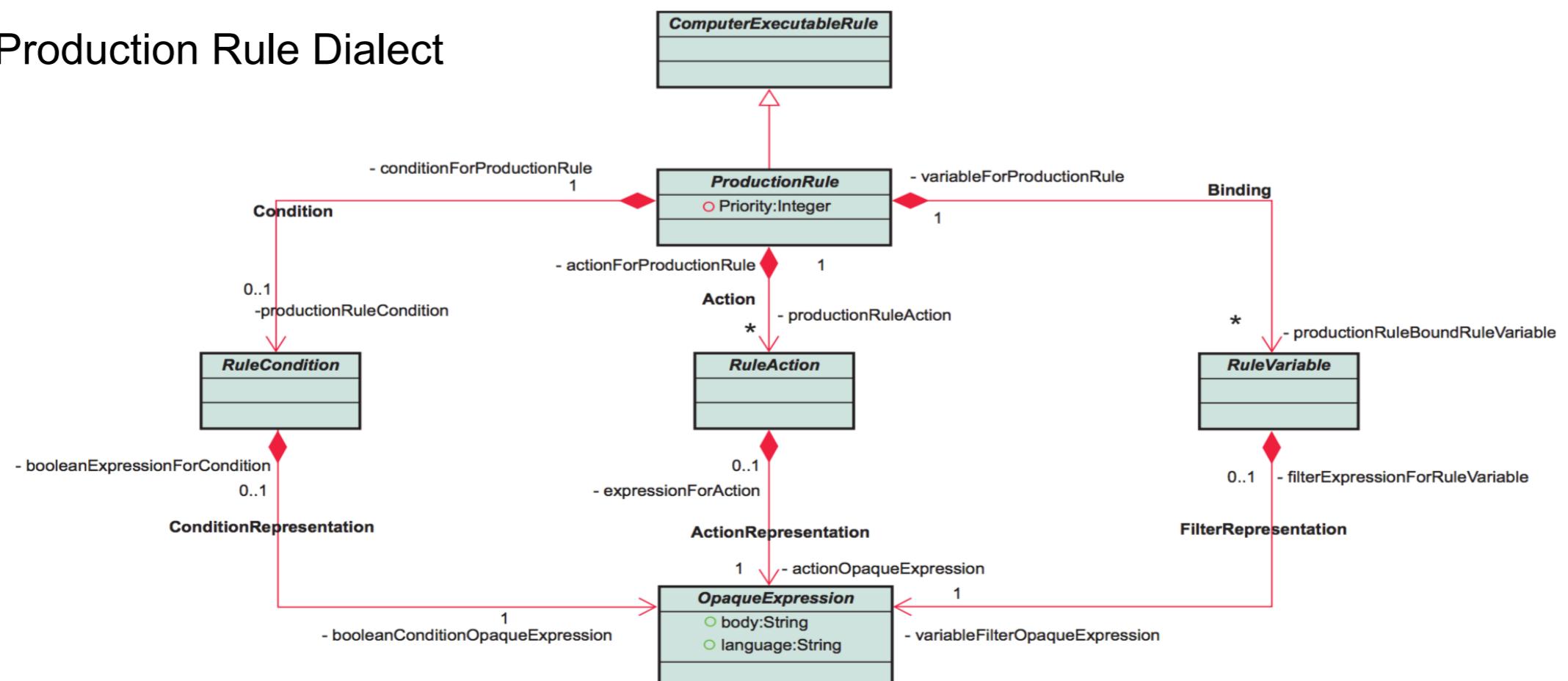
- **The execution of Production Rules is typically based on the Rete algorithm (or variants)**
 - ◆ Defined in the 1982 by Charles Forgy, for expert systems
 - ◆ Builds a network of nodes, where each intermediate node corresponds to a condition pattern in the antecedent of a rule
 - root nodes contains classes
 - leafs contain actions
 - there are other intermediate nodes for conjunctions of conditions (reused for several actions/rules)
 - ◆ (new) facts are matched against these patterns (incremental evaluation)
 - ◆ Nodes internally store information about its satisfaction
 - ◆ Once a node is totally satisfied, the corresponding action(s) is triggered
- **Order conflicts between actions are solved by several criteria: priority, recency, specificity, refraction, ...**

Production Rules Execution

- **The execution of Production Rules is typically based on the Rete algorithm (or variants)**
 - ◆ Defined in the 1982 by Charles Forgy, for expert systems
 - ◆ Builds a network of nodes, where each intermediate node corresponds to a condition pattern in the antecedent of a rule
 - root nodes contains classes
 - leafs contain actions
 - there are other intermediate nodes for conjunctions of conditions (reused for several actions/rules)
 - ◆ (new) facts are matched against these patterns (incremental evaluation)
 - ◆ Nodes internally store information about its satisfaction
 - ◆ Once a node is totally satisfied, the corresponding action(s) is triggered
- **Order conflicts between actions are solved by several criteria: priority, recency, specificity, refraction, ...**
- **Rete is also the inspiration of most CEP languages**

Production Rules Standards

- Production rules are usually “sold” as business decision support systems
- There are standards defined by standardisation institutions for the interchange of these rules
 - ◆ OMG Production Rule Representation (OMG PRR)
 - ◆ W3C RIF Production Rule Dialect



Active Rules

- Active Rules extend Production Rules with an explicit Event part. They have the form
 - ◆ **on Event if Condition do Action**

Active Rules

- Active Rules extend Production Rules with an explicit Event part. They have the form
 - ◆ **on *Event* if *Condition* do *Action***
- They have their roots on active databases (and triggers)

Active Rules

- Active Rules extend Production Rules with an explicit Event part. They have the form
 - ◆ *on Event if Condition do Action*
- They have their roots on active databases (and triggers)
- Events and condition testing is separate

Active Rules

- Active Rules extend Production Rules with an explicit Event part. They have the form
 - ◆ *on Event if Condition do Action*
- They have their roots on active databases (and triggers)
- Events and condition testing is separate
- Events may be atomic, or complex, defined by event algebras

Active Rules

- Active Rules extend Production Rules with an explicit Event part. They have the form
 - ◆ *on Event if Condition do Action*
- They have their roots on active databases (and triggers)
- Events and condition testing is separate
- Events may be atomic, or complex, defined by event algebras
- Actions may include raising of other events
 - ◆ This allows one to define derived events

Rewerse MARS

- Modular Active Rules for the Semantic Web
- General schema, using heterogeneous languages, for reactivity on changes in (semantic) web resources. Includes brokers for
 - ◆ Event languages (e.g. SNOOP)
 - ◆ Query language (SPARQL, OWLQ)
 - ◆ Test languages (e.g. propositional and 1st order logic)
 - ◆ Action languages (e.g. SPARQL-update, CCS and several imperative languages)
- Rules are written in OWL

- **Extension of XML update language XChange, to deal with XML events**
 - ◆ on Event Query if XML query do XML update
 - ◆ The event query specifies events in XML format, issued by web nodes (with emission and reception time)
 - ◆ The query part is an Xcerpt query over an XML document
 - ◆ The action part is an XChange update command over an XML document
- **Complex events are expressed as compound queries to received XML events**
 - ◆ With temporal restrictions and event compositions

XChange^{EQ} example

```
andthen [
  xchange:event {{
    xchange:sender {"http://airline.com"},  

    cancellation-notification {{
      flight {{ number { var Number } } } }
    }},
  xchange:event {{
    xchange:sender {"http://airline.com"},  

    important {"Accomodation is not granted!"}
  }}
] within 2 h
```

IBM WebSphere Business Events

- High level language for defining business processes that may react on

The screenshot shows the configuration interface for an 'Interaction Set' named 'Process Delivery Bids'. The interface is divided into two main sections: 'In response to:' and 'Else'.

In response to:

- Delivery Bid (Driver System)

Else:

- Immediately
 - If: Store does Manual Assignment
 - Then: Log Delivery Bid (Scheduling System)
 - Else: no action
- Immediately
 - If: Store does Automatic Assignment and Delivery not yet Assigned and Committed Time is Satisfactory and delivery was not cancelled
 - Then: Execute Assignment (Scheduling System)
 - Else: no action

More active languages

■ ruleCore

- ◆ XML based reactive language, with SNOOP based complex event detection

■ Reaction RuleML

- ◆ Rule interchange format for active rules
- ◆ Includes support for complex events, with a great variety of algebra operators, different types of event detection and consumption, different semantics of events (time and interval based), ...

Logic Programming

- Declarative (deduction) rules used for programming but also for representing (static) knowledge

$H :- A_1, \dots, A_m, \neg B_1, \dots, \neg B_n$

- Non-monotonic language (with negation by default) for representing common sense knowledge

Logic Programming

- Declarative (deduction) rules used for programming but also for representing (static) knowledge

$H :- A_1, \dots, A_m, \neg B_1, \dots, \neg B_n$

- Non-monotonic language (with negation by default) for representing common sense knowledge
- It was noted (in the late 80s) that LP was well suited also for representing knowledge that changes over time
 - ◆ Frame problem: How to state that, things *normally* stay the same?
 - ◆ When an action is performed, the truth of a statement is kept, unless there is something that changes it (i.e. if it is false by default that something changes it)

Event Calculus

- A methodology to represent in LP knowledge that changes due to the occurrence of events
- Basic concepts
 - ◆ Fluents are true at time points
 - ◆ The truth of fluents is represented by `holds_at(Fluent, Time)`
 - ◆ The occurrence of (raw) events is signalled by `occurs(Event, T)`
 - ◆ The effect of event is expressed by `initiates(Event, Fluent, Time)` and `terminates(Event, Fluent, Time)`
 - ◆ Predicate `</2` for ordering time points.
- The semantics is given by a logic program that defines `holds_at/2` in terms of the other predicates

Event Calculus Semantics

■ The definition of `holds_at`

```
holds_at(F,T) :- occurs(E,T1), initiates(E,F,T1),  
                  T1 < T, not clipped(F,T1,T).
```

```
clipped(F,T1,T2) :- occurs(E,T), T1 ≤ T < T2,  
                     terminates(E,F,T).
```

■ Examples:

```
initiates(stopService,serviceUnavailable,T).
```

```
terminates(startService,serviceUnavailable,T).
```

```
occurs(e1,T) :- occurs(e2,T), occurs(e3,T).
```

```
occurs(e1,T) :- occurs(e2,T1), occurs(e3,T), T1 ≤ T.
```

Transaction Logic (TR)

- Instead of using LP for representing a dynamic world with meta-predicates, extend it to deal with time
- Transaction logic was defined in the mid 90s to express (database) transactions. Extends LP with rules of the form

$H :- A_1 \otimes \dots \otimes A_n$

- Formulas are true in paths (rather than states)
 - ◆ The rule reads, H , is true in a path where A_1 is true in the beginning, and then ... and then A_n is true
- Besides \otimes , these rules can also have “,” to denote truth at the same time

TR for event processing

- It is very easy to define complex events

wrongWay(Id) :-

segSpeed(Id , Seg , _) \otimes segSpeed(Id , Seg-1 , _) .

- TR has been extended in several ways to deal with several event operators
 - ◆ ETALIS includes several event operators, including with (real) time constraints
 - ◆ TR^{ev} includes mechanisms to simultaneously deal with complex events and ACID transactions

TR for event processing

- **It is very easy to define complex events**

wrongWay(Id) :-

segSpeed(Id , Seg , _) \otimes segSpeed(Id , Seg-1 , _) .

- **TR has been extended in several ways to deal with several event operators**

- ◆ ETALIS includes several event operators, including with (real) time constraints
- ◆ TR^{ev} includes mechanisms to simultaneously deal with complex events and ACID transactions

- **But these LP-based languages have much more features, to represent knowledge, and to perform elaborate reasoning!**

- ◆ Reasoning over represent knowledge, and streaming data (events)

Example: Acting upon traffic violations

A system for the police to monitor, detect and act upon traffic violations based on a sensor network in roads

- A sensor identifies plates and distinguish between types of vehicles
- Based on the info from sensors, and about vehicles, it must reason about vehicles' traffic violations
- For cars violating traffic rules, it must issue fines and notify the drivers

Required Ingredients

- The sensor network (of course!)
- Data published by the sensors in some processable way
 - **E.g. publish data in RDF**
- Combining data from different types of sensors, for interoperability
 - **Semantic Sensor Web ontologies**
- Detecting complex events, based on atomic data from sensors
 - **Complex event detection languages**
- Reasoning with sensor data together with data about vehicles, vehicles types, kinds of violations, etc
 - **Ontologies (e.g. in OWL)**
- Expressing rules describing how to act upon traffic violations
 - **Event-Condition-Action rules**

Example in \mathcal{TR}^{ev}

Application specific RDF data

ov : vehicle	rdf : type	owl : Class .
ov : motorVehicle	rdfs : subClassOf	ov : vehicle .
ov : lightVehicle	rdfs : subClassOf	ov : motorVehicle .
ov : heavyVehicle	rdfs : subClassOf	ov : vehicle .
ov : sensor	rdf : type	owl : Class .
ov : sensor1	rdf : type	ov : sensor .
ov : sensor2	rdf : type	ov : sensor .

Definition of transactions

```
processViolation(P, DT, V) ← fineCost(V, Cost) ⊗ isDriver(P, D)⊗  
    fineIssued(P, D, DT, Cost).ins ⊗ notifyFine(P, D, DT, Cost)
```

```
notifyFine(P, D, DT, Cost) ← hasAddress(D, Addr)⊗  
    sendLetter(D, Addr, P, DT, Cost)
```

Example in \mathcal{TR}^{ev}

Event Detection

```
o(passingWrongWay(P, DT1)) ←  
  [o((Obs1 ov:plateRead P).ins)  
   ∧ o((Obs1 ov:vehicleDetected motorVehicle).ins)  
   ∧ o((Obs1 ov:dateTime DT1).ins) ∧ o((Obs1 ov:readBy sensor2).ins)]  
  ⊗  
  [o((Obs2 ov:plateRead P).ins)  
   ∧ o((Obs2 ov:vehicleDetected motorVehicle).ins)  
   ∧ o((Obs2 ov:dateTime DT2).ins) ∧ o((Obs2 ov:readBy sensor1).ins))]  
   ∧ (DT1 < DT2)
```

Event Response

```
r(passingWrongWay(P, DT)) ← processViolation(P, DT, wrongWay)
```

Behaviour in a concrete situation

ov : obs ₁	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213000 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor1 .
ov : obs ₂	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213516 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor2 .

Behaviour in a concrete situation

ov : obs ₁	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213000 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor1 .
ov : obs ₂	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213516 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor2 .

- Since, from the ontology, `heavyVehicle ⊑ vehicle`,
`o((ov:obs1 ov:vehicleDetected motorVehicle).ins)` holds in the same transition as `o((ov:obs1).ins)`

Behaviour in a concrete situation

ov : obs ₁	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213000 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor1 .
ov : obs ₂	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213516 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor2 .

- Similarly **o((ov:obs₂ ov:vehicleDetected motorVehicle).ins)** holds together with **o((ov:obs₁).ins)**

Behaviour in a concrete situation

ov : obs ₁	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213000 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor1 .
ov : obs ₂	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213516 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor2 .

- So, **o**(passingWrongWay("01-01-AA", 1426325213000) holds in the same transition where actions (ov:obs₁).ins \otimes (ov:obs₂).ins occur

Behaviour in a concrete situation

ov : obs ₁	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213000 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor1 .
ov : obs ₂	rdf : type	ov : Observation ;
	ov : plateRead	"01-01-AA" ;
	ov : dateTime	1426325213516 ;
	ov : vehicleDetected	ov : heavyVehicle ;
	ov : readBy	ov : sensor2 .

- Thus $(ov:obs_1).ins \otimes (ov:obs_2).ins$ only succeeds in a path where the driver of vehicle "01-01-AA" is fined and notified for the infraction of passing the road in the wrong way

Further Reading and Summary



Q&A

Further Reading

■ Recommend Readings

- ◆ Opher Etzion and Peter Niblett. Event Processing in Action. Manning Publications, 2010.

■ Supplemental readings:

- ◆ SiddhiQL Guide 3.1
 - <https://docs.wso2.com/display/CEP420/Introduction+to+CEP>
- ◆ Lukasz Golab and Tamer Özsü. Data Stream Management. Morgan and Claypool, 2010.

Further Reading and Summary



Q&A