Data Stream Management and Complex Event Processing in Esper

INF5100, Autumn 2010

Jarle Søberg

ifi

UNIVERSITY OF OSLO

Outline

- Overview of Esper
- DSMS and CEP concepts in Esper
  - Examples taken from the documentation
    - A lot of possibilities
    - We focus at the ones that extend traditional SQL
- Esper's query processing model
- Code example taken from the documentation
  - Neither the concept descriptions nor the code example are complete
  - Consult with the documentation for more details
    - http://esper.codehaus.org/esper-3.5.0/doc/reference/en/pdf/esper_reference.pdf

© Jarle Søberg, 2010

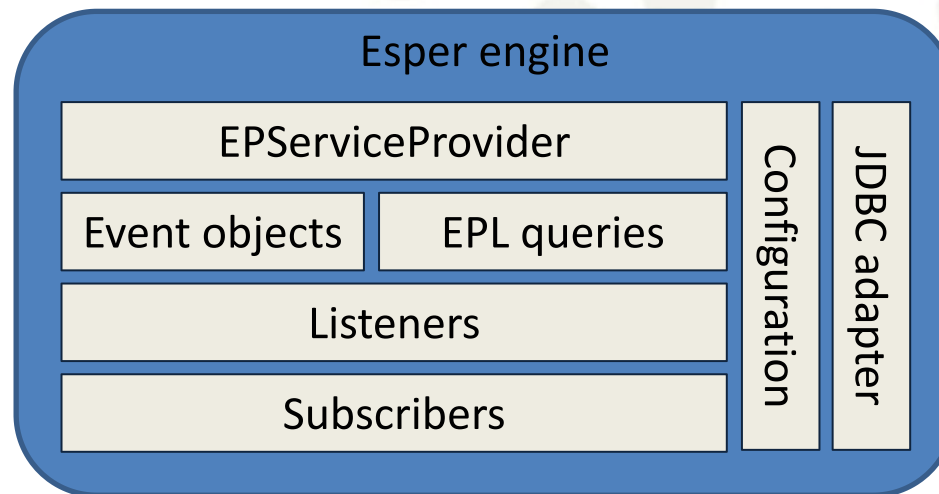
Overview of Esper

- A Java/.NET library for complex event processing
  - Stream based
  - Our examples are from the Java version
- Processes complex queries written in a language called EPL (event processing language)
  - Uses DBMS, DSMS and CEP concepts
- The code is available at esper.codehaus.org
    - Tutorial(s), code examples
- Open source
- Can be used in several data stream and CEP applications

© Jarle Søberg, 2010

Architecture Overview

- The main elements in the Esper architecture resemble any DSMS/CEP system
  - Input data is processed by one or more queries



© Jarle Søberg, 2010

Data Model (2)

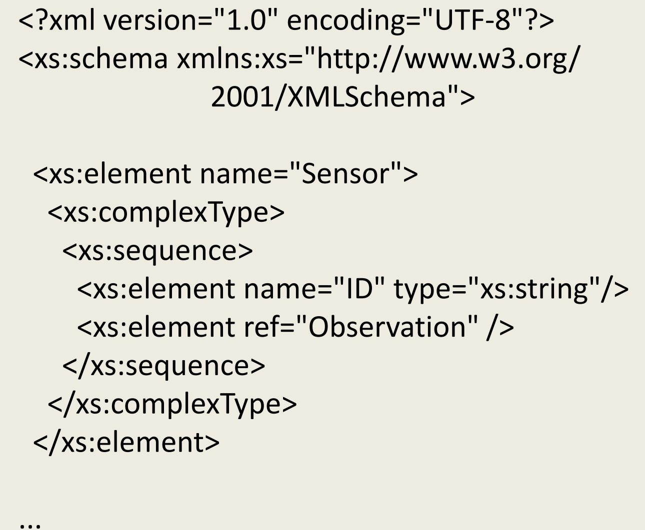
The parenthesis denote the section/chapter in the documentation where this is described.

- A data stream of event objects
  - Attributes that are defined with types and values
    - String, boolean, integer, long, float, byte, ...
    - One needs to write a method that converts to event objects from the data stream or file
  - Represented as Java objects
- Stream types
  - Input stream: `istream`
    - All the new events arriving, and entering a data window or aggregation
    - The default stream type
  - Remove stream: `rstream`
    - All the old events leaving a data window or aggregation
  - Both: `irstream`

© Jarle Søberg, 2010

Event Declaration (2)

- The event objects can be declared by
  - Schema
    - **`create schema`** ***`schema_name [as] (property_name property_type [,property_name property_type [,...]) `*** **`[inherits`** ***`inherited_event_type[, inherited_event_type] [,...]]`***
  - XML
    - Events can be represented as **`org.w3c.dom.Node`** instances



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/
                2001/XMLSchema">

 <xs:element name="Sensor">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="ID" type="xs:string"/>
    <xs:element ref="Observation" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>

...
```

  - Code

```
package org.myapp.event;

public class OrderEvent {
    private String itemName;
    private double price;

    public OrderEvent(String itemName, double price) {
        this.itemName = itemName;
        this.price = price;
    }

    public String getItemName() {
        return itemName;
    }

    public double getPrice() {
        return price;
    }
}
```

  - Many other types
    - E.g. **`java.util.Map`**

© Jarle Søberg, 2010

Creating Events

- Create a new event object and send the object to the event processor runtime:

```
OrderEvent event =
     new OrderEvent("shirt", 74.50);
epService.getEPRuntime().
     sendEvent(event);
```

© Jarle Søberg, 2010

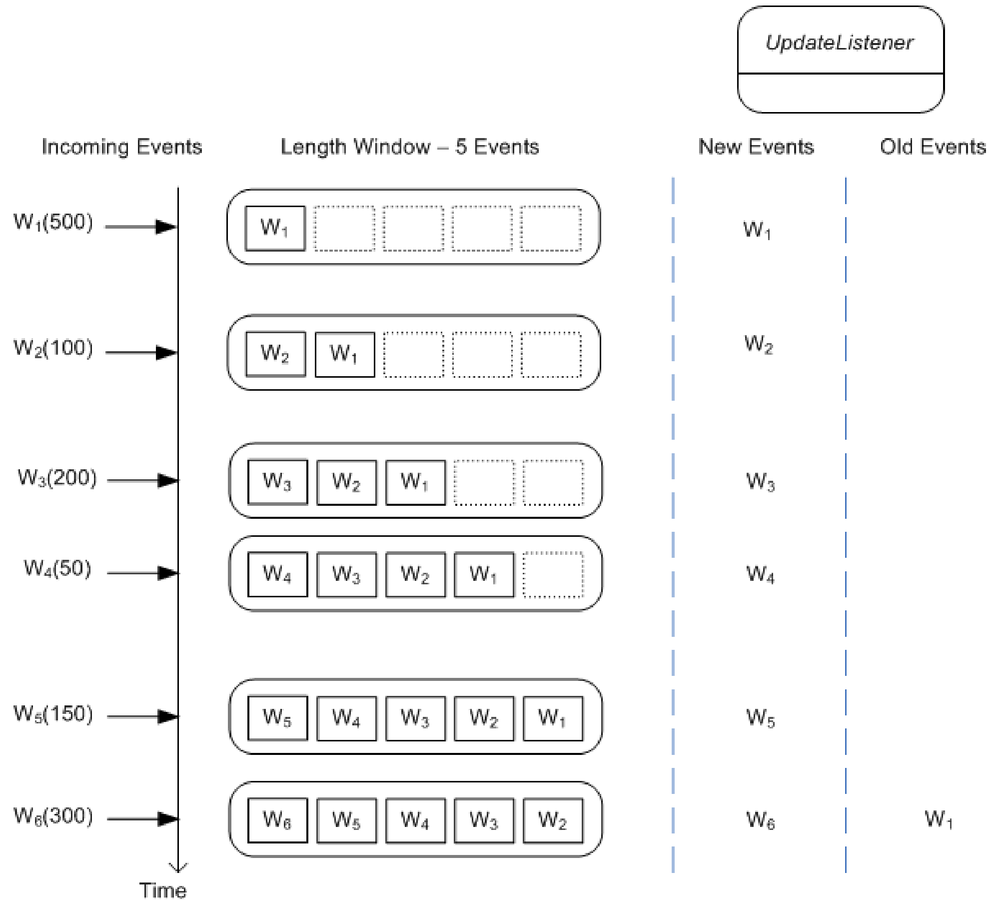
Windows (9)

- In Esper a window is called a *view*
- Provides many types of windows/views
  - some examples:

View	Syntax	Description
Length window	win:length(*size)*	Sliding window by number of elements
Length batch window	win:length_batch(*size)*	Tumbling window up to *size* events
Time window	win:time(*time period)*	Sliding time window
Externally-timed window	win:ext_timed(*timestamp expression, time period)*	Sliding time window, based on the millisecond time value supplied by an expression
Time batch window	win:time_batch(*time period[,optional reference point] [, flow control])*	Tumbling window that batches events and releases them every specified time interval, with flow control options

© Jarle Søberg, 2010

Windows - Example



© Jarle Søberg, 2010

UNIVERSITY OF OSLO

Timestamps and Time Specification

- Implicit timing
  - Esper uses the system clock
- Explicit timing
  - Can use the timestamps on the data in the data stream
- Defines time periods in the queries

time-period : [*day-part] [hour-part] [minute-part] [seconds-part] [milliseconds-part]*
day-part : *(number|variable_name) ("days" | "day")*
hour-part : *(number|variable_name) ("hours" | "hour")*
minute-part : *(number|variable_name) ("minutes" | "minute" | "min")*
seconds-part : *(number|variable_name) ("seconds" | "second" | "sec")*
milliseconds-part : *(number|variable_name) ("milliseconds" | "millisecond" | "msec")*

Aggregations (4.6)

- The aggregate functions are **`sum`**, **`avg`**, **`count`**, **`max`**, **`min`**, **`median`**, **`stddev`**, and **`avedev`**
- For example, to find out the total price for all stock tick events in the last 30 seconds

```
select sum(price) from
StockTickEvent.win:time(30 sec)
```

- Supports group-by and having

© Jarle Søberg, 2010

Output Definitions (4.7)

- Optional
  - control or stabilize the rate at which events are output and to suppress output events
  - Examples
    - `select sum(price) from OrderEvent.win:time(30 min) output snapshot every 60 seconds`
    - `select * from StockTickEvent.win:time(30 sec) output every 5 events`
    - `select * from StockTickEvent.win:length(5) output every 1.5 minutes`

© Jarle Søberg, 2010

Esper Syntax for Data Stream Management (4.2)

- Basic SQL-inspired queries for data stream management called EPL

```
[insert into insert_into_def]
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

- DBMS support for joins between databases and data streams
  - Historical queries
  - Additional information
    - E.g. location of a sensor

© Jarle Søberg, 2010

Patterns for Complex Event Processing (5)

- Extends Esper beyond windows and aggregations (DSMS features)
- Uses atoms, e.g. expressions
- Operators
  - Control pattern sub-expression repetition: **`every`**, **`every-distinct`**, **`[num]`** and **`until`**
  - Logical operators: **`and`**, **`or`**, **`not`**
  - Temporal operators that operate on event order: **`->`** (followed-by)
  - Guards control the lifecycle of sub-expressions, e.g. **`timer:within`**, **`timer:withinmax`** and **`while`**-expression.
    - Custom plug-in guards may also be used.
- Example of patterns

```
select a.custId, sum(a.price + b.price)
from pattern [every a=ServiceOrder ->
b=ProductOrder(custId = a.custId) where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

© Jarle Søberg, 2010

The `every` Operator (5.5.1)

- Tells if every expression should be investigated
- Example stream: A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4
  - every (A -> B)
    - Matches on B_1 for combination $\{A_1, B_1\}$
    - Matches on B_3 for combination $\{A_3, B_3\}$
    - Matches on B_4 for combination $\{A_4, B_4\}$

© Jarle Søberg, 2010

The **every** Operator (cont.)

- Example stream: A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4
  - every A -> B
    - Matches on B_1 for combination {A_1, B_1}
    - Matches on B_3 for combination {A_2, B_3} and {A_3, B_3}
    - Matches on B_4 for combination {A_4, B_4}
  - A -> every B
    - Matches on B_1 for combination {A_1, B_1}.
    - Matches on B_2 for combination {A_1, B_2}.
    - Matches on B_3 for combination {A_1, B_3}
    - Matches on B_4 for combination {A_1, B_4}
  - every A -> every B
    - Matches on B_1 for combination {A_1, B_1}.
    - Matches on B_2 for combination {A_1, B_2}.
    - Matches on B_3 for combination {A_1, B_3} and {A_2, B_3} and {A_3, B_3}
    - Matches on B_4 for combination {A_1, B_4} and {A_2, B_4} and {A_3, B_4} and {A_4, B_4}

© Jarle Søberg, 2010

Query Processing Model in Esper

1. Sources
   - Data tuples from sensors, trace files, etc.
   - Push based
     - A method pulls data from the source and pushes the data to the query processor
2. Queries
   - Process data tuples
   - Push based
3. Listeners
   - Receive data tuples from queries
   - Push data tuples to other queries
4. Subscribers
   - Receive processed data tuples

- A set of queries and listeners form graphs during runtime with the data streams as input and the subscribers as output
- These graphs are manually created by the application programmer

© Jarle Søberg, 2010

Query Processing Model in Esper (cont.)

Example:

Subscriber

Listener

Listener

Query

Event stream

Query

Event stream

Listener

Event stream

Listener

Query

Listener

Query

Query

Event stream

Sensors/Trace files

© Jarle Søberg, 2010

Performance

- Esper exceeds over 500 000 event/s on a dual CPU 2GHz Intel based hardware, with engine latency below 3 microseconds average
  - Demonstrates linear scalability from 100 000 to 500 000 event/s
  - Consistent results across different queries
  - The internal data structures are optimized for minimal locking and high write speed
  - Esper is fully multi-thread safe
  - Esper internally builds all the indexes and uses many optimization techniques hidden to the application
- Currently *unknown* what Esper will do in an overload situation
  - No sampling techniques are described in the documentation

© Jarle Søberg, 2010

Code Example

- Esper is not only query writing
  - Needs setup of listeners, event descriptions, etc.
  - We go through a representative and good example taken from the Esper tutorial
- A running version of Esper is provided to all the groups in the mandatory assignment
  - There you will see how the listeners and queries are configured and work
- The following slides show some highlights

© Jarle Søberg, 2010

Code Example: The Query

- Registering the query that returns the average price over all OrderEvent events that arrived in the last 30 seconds:

```
EPServiceProvider epService =
  EPServiceProviderManager.getDefaultProvider()
  ;
String expression = "select avg(price) from
  org.myapp.event.OrderEvent.win:time(30 sec)";
EPStatement statement =
  epService.getEPAdministrator().
   createEPL(expression);
```

© Jarle Søberg, 2010

Code Example: Adding the Listener

- Register a listener that listens to the query

```
public class MyListener implements
  UpdateListener {
    public void update(EventBean[] newEvents,
      EventBean[] oldEvents) {
        EventBean event = newEvents[0];
        System.out.println("avg=" +
          event.get("avg(price)"));
    }
}
/*---*/
MyListener listener = new MyListener();
statement.addListener(listener);
```

This has to be similar to the projected attributes and aggregations in the query!

© Jarle Søberg, 2010

Code Example: Configuration

- Finally, some configuration has to be done:

```
Configuration config = new
 Configuration();

config.
 addEventTypeAutoAlias("org.myapp.even
 t");

EPServiceProvider epService =
 EPServiceProviderManager.
     getDefaultProvider(config);
```

© Jarle Søberg, 2010

Conclusion

- Esper is a general and easy-to-use library for data stream management and complex event processing
  - Implements many of the DSMS/CEP concepts like windowing, aggregations, and patterns
- Esper will be used in the mandatory assignment
  - Knowledge from this introduction needs to be extended by the documentation in the Esper pages
    - http://esper.codehaus.org/esper-3.5.0/doc/reference/en/pdf/esper_reference.pdf
  - Some of the configuration parts of the mandatory assignment have been simplified so that you do not have to spend too much time on that when doing the mandatory assignment

© Jarle Søberg, 2010