Esper Reference Documentation

Version: 0.7.0a

Table of Contents

Preface

Analyzing and reacting to information in real-time oftentimes requires the development of custom applications. Typically these applications must obtain the data to analyze, filter data, derive information and then indicate this information through some form of presentation or communication. Data may arrive with high frequency requiring high throughput processing. And applications may need to be flexible and react to changes in requirements while the data is processed. Esper is an event stream processor that aims to enable a short development cycle from inception to production for these types of applications.

Esper is a 100% Java component that can be embedded in Java applications. It allows push and pull of data via it's a subscription and pull API. Esper can be extended by building custom views, functions, windows etc..

1. Read Section 1.1, “Introduction to CEP and event stream analysis” if you are new to CEP and ESP (complex event processing, event stream processing)
2. Read Section 5.1, “Event Pattern Overview” for an overview over event patterns
3. Read Section 6.1, “EQL Introduction” for an introduction to event stream processing via EQL
4. Then glance over the examples Section 9.1, “Overview”

Chapter 1. Technology Overview

1.1. Introduction to CEP and event stream analysis

The Esper engine has been developed to address the requirements of applications that analyze and react to events. Some typical examples of applications are:

- Business process management and automation (process monitoring, BAM, reporting exceptions)
- Finance (algorithmic trading, fraud detection, risk management)
- Network and application monitoring (intrusion detection, SLA monitoring)
- Sensor network applications (RFID reading, scheduling and control of frabrication lines, air traffic)

What these applications have in common is the requirement to process events (or messages) in real-time or near real-time. This is sometimes referred to as complex event processing (CEP) and event stream analysis. Key considerations for these types of applications are throughput, latency and the complexity of the logic required.

- High throughput - applications that process large volumes of messages (between 1,000 to 100k messages per second)
- Low latency - applications that react in real-time to conditions that occur (from a few milliseconds to a few seconds)
- Complex computations - applications that detect patterns among events (event correlation), filter events, aggregate time or length windows of events, join event streams, trigger based on absence of events etc.

The Esper engine was designed to make it easier to build and extend CEP applications.

1.2. CEP and relational databases

Relational databases and the standard query language (SQL) are designed for applications in which most data is fairly static and complex queries are less frequent. Also, most databases store all data on disks (except for in-memory databases) and are therefore optimized for disk access.

To retrieve data from a database an application must issue a query. If an application need the data 10 times per second it must fire the query 10 times per second. This does not scale well to hundreds or thousands of queries per second.

Database triggers can be used to fire in response to database update events. However database triggers tend to be slow and often cannot easily perform complex condition checking and implement logic to react.

In-memory databases may be better suited to CEP applications then traditional relational database as they generally have good query performance. Yet they are not optimized to provide immediate, real-time query results required for CEP and event stream analysis.

1.3. The Esper engine for CEP

The Esper engine works a bit like a database turned upside-down. Instead of storing the data and running queries against stored data, the Esper engine allows applications to store queries and run the data through. Response from the Esper engine is real-time when conditions occur that match queries. The execution model is thus continuous rather then only when a query is submitted.

Esper provides two principal methods or mechanisms to process events: event patterns and event stream queries.

Esper offers an event pattern language to specify expression-based event pattern matching. Underlying the pattern matching engine is a state machine implementation. This method of event processing matches expected sequences of presence or absence of events or combinations of events. It includes time-based correlation of events.

Esper also offers event stream queries that address the event stream analysis requirements of CEP applications. Event stream queries provide the windows, aggregation, joining and analysis functions for use with streams of events. These queries are following the EQL syntax. EQL has been design for similarity with the SQL query language but differs from SQL in it's use of views rather then tables. Views represent the different operations needed to structure data in an event stream and to derive data from an event stream.

Esper provides these two methods as alternatives through the same API.

Chapter 2. Architecture

2.1. Overview

A (very) high-level view of the architecture: TODO

2.2. Building and Testing

The Esper code base constists of about 300 source code and 270 unit test or test support classes, excluding examples. After build there are over 500 unit test methods that are automatically run to verify the build.

Esper requires the following 3rd-party libraries:

- ANTLR is the parser generator used for parsing and parse tree walking of the pattern and EQL syntax. Credit goes to Terence Parr at http://www.antlr.org. The ANTLR license is in the lib directory. The library is required for compile-time only.
- CGLIB is the code generation library for fast method calls. This open source software is under the Apache license. The Apache 2.0 license is in the lib directory.
- LOG4J and Apache commons logging are logging components. This open source software is under the Apache license. The Apache 2.0 license is in the lib directory.
- BeanUtils is a JavaBean manipulation library. This open source software is under the Apache license. The Apache 2.0 license is in the lib directory.
- JUnit is a great unit testing framework. It's license has also been placed in the lib directory. The library is required for build-time only.

Chapter 3. Configuration

TODO

3.1. Programmatic configuration

An instance of TODO

Chapter 4. API Reference

4.1. Overview

Esper has 2 primary interfaces that this section outlines: The admininistrative interface and the runtime interface.

Use Esper's administrative interface to create event patterns and EQL statements as discussed in Section 5.1, “Event Pattern Overview” and Section 6.1, “EQL Introduction”.

Use Esper's runtime interface to send events into the engine, emit events and get statistics for an engine instance.

The JavaDoc documentation is also a great source for API information.

4.2. Engine Instances

Each instance of an Esper engine is completely independent of other engine instances and has it's own administrative and runtime interface.

An instance of the Esper engine is obtained via static methods on the `EPServiceProviderManager` class. The `getDefaultProvider` method and the `getProvider(String URI)` methods return an instance of the Esper engine. The latter can be used to obtain multiple instances of the engine for different URI values. The `EPServiceProviderManager` determines if the URI matches all prior URI values and returns the same engine instance for the same URI value. If the URI has not been seen before, a new Engine instance is created.

The code snipped below gets the default instance Esper engine. Subsequent calls to get the default engine instance return the same instance.

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

This code snippet gets an Esper engine for uri `RFIDProcessor1`. Subsequent calls to get an engine with the same uri return the same instance.

```
EPServiceProvider epService = EPServiceProviderManager.getProvider("RFIDProcessor1");
```

A existing Esper engine instance can be reset via the `initialize` method on the `EPServiceProvider` instance. This stops and removes all statements in the Engine.

4.3. The Administrative Interface

Create event patterns or EQL statements via the administrative interface `EPAdministrator`.

This code snippet gets an Esper engine then creates an event pattern and an EQL statement.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPAdministrator admin = epService.getEPAdministrator();
EPStatement 10secRecurTrigger = admin.createPattern("every timer:at(*, *, *, *, *, */10)");
EPStatement weightedAvgView = admin.createEQL(
    "select * from MarketDataBean(symbol='IBM').win:time(60).stat:weighted_avg('price', 'volume')");
```

The `createPattern` and `createEQL` methods return `EPStatement` instances. Statement are automatically started and active when created. A statement can also be stopped and started again via the `stop` and `start` methods shown in the code snippet below.

```
weightedAvgView.stop();
weightedAvgView.start();
```

We can subscribe to updates posted by a statement via the `addListener` and `removeListener` methods the `EPStatement` statement. We need to provide an implementation of the `UpdateListener` interface to the statement.

```
UpdateListener myListener = new MyUpdateListener(); // MyUpdateListener implements UpdateListener
weightedAvgView.addListener(myListener);
```

EQL statements and event patterns publish old data and new data to registered `UpdateListener` listeners. Old data published by views constists of the events representing the prior values of derived data held by the statement. New data published by views is the events representing the new values of derived data held by the statement.

Subscribing to events posted by a statement is following a push model, ie. the engine pushes data to listeners when when events are received that cause data to change or patterns to match. Alternatively, statements can also serve up data in a pull model via the `iterator` method. This can come in handy if we are not interested in all new updates, but only want to perform a frequent poll for the latest data. For example, a event pattern that fires avery 5 seconds could be used to pull data from an EQL statement.The code snippet below demonstrates some pull code.

```
Iterator<EventBean> eventIter = weightedAvgView.iterator();
for (EventBean event : eventIter) {
   // .. do something ..
}
```

This is a second example:

```
double averagePrice = (Double) eqlStatement.iterator().next().get("average");
```

4.4. The Runtime Interface

The `EPRuntime` interface is used to send events for processing into an Esper engine, and to emit Events from an engine instance to the outside world.

The below code sippet shows how to send events to the engine.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPRuntime runtime = epService.getEPRuntime();

// Send an example event containing stock market data
runtime.sendEvent(new MarketDataBean('IBM', 75.0));
```

Another important method in the runtime interface is the `route` method. This method is designed for use by `UpdateListener` implementations that need to send events into an engine instance.

The `emit` and `addEmittedListener` methods can be used to emit events from a runtime to a registered set of one or more emitted event listeners. Events are emitted on an event channel identified by a name. Listeners are implementations of the `EmittedListener` interface. Listeners can specify a channel to listen to and thus only receive events posted to that channel. Listeners can also supply no channel name and thus receive emitted

events posted on any channel. Channels are uniquely identified by a string channel name.

4.5. Event Class Requirements

A event is a immutable record of a past occurance of an action or state change. A event can have a set of event properties that supply information about the event. A event also has an event class.

In Esper, events are object instances that expose event properties through JavaBean-style getter methods. Events classes do not have to be fully compliant to the JavaBean specification, however for the Esper engine to obtain event properties, the required JavaBean getter methods must be present.

Classes that represent events should be made immutable. As events are recordings of a state change or action that occured in the past, the relevant event properties should not be changable. However this is not a hard requirement and the Esper engine accepts events that are mutable as well.

4.6. Time-Keeping Events

Special events are provided that can be used to control the time-keeping of an engine instance. There are two models for an engine to keep track of time. Internal clocking is when the engine instance relies on the `java.util.Timer` class for time tick events. External clocking can be used to supply time ticks to the engine. The latter is useful for testing time-based event sequences or for synchronizing the engine with an external time source.

By default, the Esper engine uses internal time ticks. This behavior can be changed by sending a timer control event to the engine as shown below.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPRuntime runtime = epService.getEPRuntime();
// switch to external clocking
runtime.sendEvent(new TimerControlEvent(TimerControlEvent.ClockType.CLOCK_EXTERNAL));

// send a time tick
long timeInMillis = System.currentTimeMillis(); // Or get the time somewhere else
runtime.sendEvent(new CurrentTimeEvent(timeInMillis));
```

4.7. Events Received from the Engine

The Esper engine posts events to registered `UpdateListener` instances ('push' method for receiving events). For many statements events can also be pulled from statements via the `iterator` method. Both pull and push supply `EventBean` instances representing the events generated by the engine or events supplied to the engine. Each `EventBean` instance represents an event, with each event being either an artificial event, composite event or an event supplied to the engine via it's runtime interface.

The `getEventType` methods supplies an event's event type information represented by an `EventType` instance. The `EventType` supplies event property names and types as well as information about the underlying object to the event.

The engine may generate artificial events that contain information derived from event streams. A typical example for artificial events are the events posted for a statement to calculate univariate statistics on an event property. The below example shows such a statement and interogates the events that the engine publishes.

```
// Derive univariate statistics on price for the last 100 market data events
String viewExpr = "select * from MarketDataBean(symbol='IBM').win:length(100).stat:uni('price')";
```

```
EPStatement priceStatsView = epService.getEPAdministrator().createEQL(viewExpr);
priceStatsView.addListener(testListener);
```

```
// Example listener code
public class MyUpdateListener implements UpdateListener
{
    public void update(EventBean[] newData, EventBean[] oldData)
    {
                // Interrogate events
                System.out.println("new average price=" + newData[0].get("average");
        }
}
```

Composite events are events that aggregate one or more other events. Composite events are typically created by the engine for statements that join two event streams, and for event patterns in which the causal events are retained and reported in a composite event. The example below shows such an event pattern.

```
// Look for a pattern where AEvent follows BEvent
String pattern = "a=AEvent() -> b=BEvent";
EPStatement stmt = epService.getEPAdministrator().createPattern(pattern);
stmt.addListener(testListener);
```

```
// Example listener code
public class MyUpdateListener implements UpdateListener
{
    public void update(EventBean[] newData, EventBean[] oldData)
    {
                System.out.println("a event=" + newData[0].get("a").getUnderlying());
                System.out.println("b event=" + newData[0].get("b").getUnderlying());
        }
}
```

Chapter 5. Event Pattern Reference

5.1. Event Pattern Overview

Event patterns match when an event or multiple events occur that match the pattern's definition. Patterns can also be time-based.

Pattern expressions can consist of filter expressions combined with pattern operators. Expressions can contain further nested pattern expressions by including the nested expression(s) in `()` round brackets.

There are 5 types of operators:

1. Operators that control pattern finder creation and termination: `every`
2. Logical operators: `and`, `or`, `not`
3. Temporal operators that operate on event order: `->` (followed-by)
4. Guards are where-conditions that filter out events and cause termination of the pattern finder. Examples are `timer:within`.
5. Observers observe time events as well as other events. Examples are `timer:interval` and `timer:at`.

5.2. How to use Patterns

5.2.1. Pattern Syntax

The `createPattern` method on the `EPAdministrator` administrative interface creates pattern statements for the given pattern expression string.

This is an example pattern expression that matches on every `ServiceMeasurement` events in which the value of the `latency` event property is over 20 seconds, and on every`ServiceMeasurement` events in which the `success` property is false. Either one or the other condition must be true for this pattern to matches.

```
every ( spike=ServiceMeasurement(latency>20000) or error=ServiceMeasurement(success=false) )
```

The Java code to create this trigger is below.

```
EPAdministrator admin = EPServiceProviderManager.getDefaultProvider().getEPAdministrator();

String eventName = ServiceMeasurement.class.getName();

EPStatement myTrigger = admin.createPattern(
    "every ( spike=" + eventName + "(latency>20000) or error=" + eventName + "(success=false) )");
```

The pattern expression starts with an `every` operator to indicate that the pattern should fire for every matching events and not just the first matching event. Within the `every` operator in round brackets is a nested pattern expression using the `or` operator. The left hand of the `or` operator is a filter expression that filters for events with a high latency value. The right hand of the operator contains a filter expression that filters for events with error status. Filter expressions are explained in Section 5.3, “Filter Expressions”.

5.2.2. Subscribing to Pattern Events

When a pattern fires it publishes one or more events to any listeners to the pattern statement. The listener inter-

face is the `net.esper.client.UpdateListener` interface.

The example below shows an anonymous implementation of the `net.esper.client.UpdateListener` interface. We add the anonymous listener implementation to the `myPattern` statement created earlier. The listener code simply extracts the underlying event class.

```
myPattern.addListener(new UpdateListener()
{
        public void update(EventBean[] newEvents, EventBean[] oldEvents)
        {
                ServiceMeasurement spike = (ServiceMeasurement) newEvents[0].get("spike");
                ServiceMeasurement error = (ServiceMeasurement) newEvents[0].get("error");
                ... // either spike or error can be null, depending on which occured
                ... // add more logic here
        }
});
```

Listeners receive an array of `EventBean` instances in the `newEvents` parameter. There is one `EventBean` instance passed to the listener for each combination of events that matches the pattern expression. At least one `EventBean` instance is always passed to the listener.

The properties of each `EventBean` instance contain the underlying events that caused the pattern to fire, if events have been named in the filter expression via the `name=eventType` syntax. The property name is thus the name supplied in the pattern expression, while the property type is the type of the underlying class, in this example `ServiceMeasurement`.

5.2.3. Pulling Data from Patterns

Data can also be pulled from pattern statements via the `iterator()` method. If the pattern had fired at least once, then the iterator returns the last event for which it fired. The `hasNext()` method can be used to determine if the pattern had fired.

```
if (myPattern.iterator().hasNext())
{
        ServiceMeasurement event = (ServiceMeasurement) view.iterator().next().get("alert");
    ... // some more code here to process the event
}
else
{
    ... // no matching events at this time
}
```

5.3. Filter Expressions

This chapter outines how to filter events based on their properties.

The simplest form of filter is a filter for events of a given type. This filter matches any event of that type regardless of the event's properties.

```
mypackage.RfidEvent()
```

The filtering criteria to be applied to events of that type are placed within brackets.

```
mypackage.RfidEvent(category="Perishable")
```

The supported filter operators are

- equals `=`
- comparison operators `< , > , >=, <=`
- ranges use the keyword `in` and round `(...)` or square brackets `[]`

Ranges come in the following 4 varieties. The use of round `()` or square `[]` bracket dictates whether an endpoint is included or excluded.

- Open ranges that contain neither endpoint `(low, high)`
- Closed ranges that contain both endpoints `[low, high]`
- Half-open ranges that contain the low endpoint but not the high endpoint `[low, high)`
- Half-closed ranges that contain the high endpoint but not the low endpoint `(low, high]`

Filter criteria are listed in a comma-separated format. In the example below we look for `RfidEvent` events with a `grade` property between 1 and 2 (endpoints included), a `price` less then 1, and a category of "Perishable".

```
mypackage.RfidEvent(category="Perishable", price<1.00, grade in [1, 2])
```

Some limitations of filters are:

- Range and comparison operators require the event property to be of a numeric type.
- Null values in filter criteria are currently not allowed.
- Filter criteria can list the same event property only once.
- Events that have null values for event properties listed in the filter criteria do not match the criteria.

5.4. Pattern Operators

5.4.1. Every

The `every` operator indicates that the pattern expression should restart when the pattern matches. Without the `every` operator the pattern expressions matcher stops when the pattern matches once.

Thus the `every` operator works like a factory for the pattern expression contained within. When the pattern expression within it fires and thus quits checking for events, the `every` causes the start of a new pattern matcher listening for more occurances of the same event or set of events.

Every time a pattern expression within an `every` operator turns true a new active pattern matcher is started looking for more event(s) or timing conditions that match the pattern expression. If the `every` operator is not specified for an expression, the expression stops after the first match was found.

This pattern fires when encountering event A and then stops looking.

```
A()
```

This pattern keeps firing when encountering event A, and doesn't stop looking.

```
every A()
```

Lets consider an example event sequence as follows.

$A_1\ B_1\ C_1\ B_2\ A_2\ D_1\ A_3\ B_3\ E_1\ A_4\ F_1\ B_4$

Table 5.1. 'Every' operator examples

Example	Description
`every ( A() -> B() )`	Detect event A followed by event B. At the time when B occurs the pattern matches, then the pattern matcher restarts and looks for event A again. 1. Matches on B_1 for combination $\{A_1, B_1\}$ 2. Matches on B_3 for combination $\{A_2, B_3\}$ 3. Matches on B_4 for combination $\{A_4, B_4\}$
`every A() -> B()`	The pattern fires for every event A followed by an event B. 1. Matches on B_1 for combination $\{A_1, B_1\}$ 2. Matches on B_3 for combination $\{A_2, B_3\}$ and $\{A_3, B_3\}$ 3. Matches on B_4 for combination $\{A_4, B_4\}$
`A() -> every B()`	The pattern fires for an event A followed by every event B. 1. Matches on B_1 for combination $\{A_1, B_1\}$. 2. Matches on B_2 for combination $\{A_1, B_2\}$. 3. Matches on B_3 for combination $\{A_1, B_3\}$ 4. Matches on B_4 for combination $\{A_1, B_4\}$
`every A() -> every B()`	The pattern fires for every event A followed by every event B. 1. Matches on B_1 for combination $\{A_1, B_1\}$. 2. Matches on B_2 for combination $\{A_1, B_2\}$. 3. Matches on B_3 for combination $\{A_1, B_3\}$ and $\{A_2, B_3\}$ and $\{A_3, B_3\}$ 4. Matches on B_4 for combination $\{A_1, B_4\}$ and $\{A_2, B_4\}$ and $\{A_3, B_4\}$ and $\{A_4, B_4\}$

The examples show that it is possible that a pattern fires for multiple combinations of events that match a pattern expression. Each combination is posted as an `EventBean` instance to the `update` method in the `UpdateListener` implementation.

5.4.2. And

Similar to the Java && operator the `and` operator requires both nested pattern expressions to turn true before the whole expression turns true (a join pattern).

Pattern matches when both event A and event B are found.

```
A() and B()
```

Pattern matches on any sequence A followed by B and C followed by D, or C followed by D and A followed by B

```
(A() -> B()) and (C() -> D())
```

5.4.3. Or

Similar to the Java "||" operator the `or` operator requires either one of the expressions to turn true before the

whole expression turns true.

Look for either event A or event B. As always, A and B can itself be nested expressions as well.

```
A() or B()
```

Detect all stock ticks that are either above or below a threshold.

```
every (StockTick(symbol='IBM', price < 100) or StockTick(symbol='IBM', price > 105)
```

5.4.4. Not

The `not` operator negates the truth value of an expression. Pattern expressions prefixed with `not` are automatically defaulted to true.

This pattern matches only when an event A is encountered followed by event B but only if no event C was encountered before event B.

```
( A() -> B() ) and not C()
```

5.4.5. Followed-by

The followed by `->` operator specifies that first the left hand expression must turn true and only then is the right hand expression evaluated for matching events.

Look for event A and if encountered, look for event B. As always, A and B can itself be nested event pattern expressions.

```
A() -> B()
```

Pattern to match when 2 status events indicating an error occur.

```
StatusEvent(status="ERROR") -> StatusEvent(status="ERROR")
```

5.5. Guards

5.5.1. timer:within

The `timer:within` guard acts like a stopwatch. If the associated pattern expression does not turn true within the specified time period it is stopped and permanently false.

Pattern for all A events that arrive within 5 seconds.

```
every A() where timer:within (5000)
```

Pattern for any A or B events in the next 5 seconds.

```
( A() or B() ) where timer:within (5000)
```

Pattern if for any 2 errors that happen 10 seconds within each other.

```
every (StatusEvent(status="ERROR") -> StatusEvent(status="ERROR") where timer:within (10000))
```

5.6. Pattern Observers

5.6.1. timer:interval

The `timer:interval` observer takes a wait time in milliseconds and waits for the defined time before the truth value of the observer turns true.

After event A arrived wait 10 seconds then indicate that the pattern matches.

```
A() -> timer:interval(10000)
```

An expression that matches every 20 seconds.

```
every timer:interval(20000)
```

5.6.2. timer:at

The `timer:at` observer is similar in function to the Unix "crontab" command. At a specified time the expression turns true. The `at` operator can also be made to pattern match at regular intervals by using an `every` operator in front of the `timer:at` operator.

The syntax is: `timer:at (minutes, hours, days of month, months, days of week [, seconds]).`

The value for seconds is optional. Each element allows wildcard `*` values. Ranges can be specified by means of lower bounds then a colon ':' then the upper bound. The division operator `*/x` can be used to specify that every xth value is valid. Combinations of these operators can be used by placing these into square brackets([]).

This expression pattern matches every 5 minutes past the hour.

```
every timer:at(5, *, *, *, *)
```

The below at operator pattern matches every 15 minutes from 8am to 5pm on even numbered days of the month as well as on the first day of the month.

```
timer:at (*/15, 8:17, [*/2, 1], *, *)
```

Chapter 6. EQL Reference

6.1. EQL Introduction

EQL statements are used to derive and aggragate information from one or more streams of events, and to join event streams. This section outlines EQL syntax. It also outlines the built-in views, which are the building blocks for deriving and aggregating information from event streams.

EQL is similar to SQL in it's use of the `select` clause and the `where` clause. Where EQL differs most from SQL is in the use of tables. EQL replaces tables with the concept of event streams.

EQL statements contain definitions of one or more views. Similar to tables in an SQL statement, views define the data available for querying and filtering. Some views represent windows over a stream of events. Other views derive statistics from event properties, group events or handle unique event property values. Views can be staggered onto each other to build a chain of views. The Esper engine makes sure that views are reused among EQL statements for efficiency.

The built-in set of views is:

1. Views that represent moving event windows: `win:lenght`, `win:time`, `win:time_batch`, `win:ext_time`, `ext:sort_window`
2. Views for aggregation: `std:unique`, `std:group`, `std:last`,
3. Views that derive statistics: `std:size`, `stat:uni`, `stat:linest`, `stat:correl`, `stat:weighted_avg`, `stat:multidim_stat`

Esper can be extended by plugging-in custom developed views.

6.2. EQL Syntax

EQL queries are created and stored in the engine, and publish results as events are received by the engine or timer events occur that match the criteria specified in the query. Events can also be pulled from running EQL queries.

The select clause in an EQL query specifies the event properties or events to retrieve. The from clause in an EQL query specifies the event stream definitions and stream names to use. The where clause in an EQL query specifies search conditions that specify which event or event combination to search for. For example, the following statement returns the average price for IBM stock ticks in the last 30 seconds if the average hit 75 or more.

```
select average from StockTick(symbol='IBM').win:time(30).stat:uni('price') where average >= 75;
```

EQL queries follow the below syntax. EQL queries can be simple queries or more complex queries. A simple select contains only a select clause and a single stream definition. Complex EQL queries can be build that feature a more elaborate select list utilizing expressions, may join multiple streams or may contain a where clause that with search conditions.

```
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
```

6.3. Choosing Event Properties And Events: the *Select* Clause

The select clause is required in all EQL statements. The select clause can be used to select all properties via the wildcard `*`, or to specify a list of event properties and expressions. The select clause defines the event type (event property names and types) of the resulting events published by the statement, or pulled from the statement.

6.3.1. Choosing all event properties: select *

The syntax for selecting all event properties in a stream is:

```
select * from stream_def
```

The following statement selects all univariate statistics properties for the last 30 seconds of IBM stock ticks for price.

```
select * from StockTick(symbol='IBM').win:time(30).stat:uni('price')
```

In a join statement, using the `select *` syntax selects event properties that contain the events representing the joined streams themselves.

6.3.2. Choosing specific event properties

To chose the particular event properties to return:

```
select event_property [, event_property] [, ...] from stream_def
```

The following statement selects the count and standard deviation properties for the last 100 events of IBM stock ticks for volume.

```
select count, stdev from StockTick(symbol='IBM').win:length(100).stat:uni('volume')
```

6.3.3. Expressions

The select clause can contain one or more expressions.

```
select expression [, expression] [, ...]from stream_def
```

The following statement selects the volume multiplied by price for a time batch of the last 30 seconds of stock tick events.

```
select volume * price from StockTick().win:time_batch(30)
```

6.3.4. Renaming event properties

Event properties and expressions can be renamed using below syntax.

```
select [event property | expression ] as identifier [, ...]
```

The following statement selects volume multiplied by price and specifies the name *volPrice* for the event prop-

erty.

```
select volume * price as volPrice from StockTick().win:length(100)
```

6.4. Specifying Event Streams : the *From* Clause

The from clause is required in all EQL statements. It specifies one or more event streams. Each event stream can optionally be given a name by means of the `as` syntax.

```
from stream_def [as name] [, stream_def [as name]] [, ...]
```

The event stream definition *stream_def* as shown in the syntax above constists of an event type, an optional filter property list and an optional list of views that derive data from a stream must be supplied. The syntax for an event stream definition is as below:

```
event_type ( [filter_criteria] ) [.view_spec] [.view_spec] [...]
```

The following EQL statement selects all event properties for the last 100 events of IBM stock ticks for volume. In the example, the event type is the fully qualified class name `org.esper.example.StockTick`. The optional filter criteria consists of a filter for the event property `symbol` with the value of "IBM". The optional view specifications for deriving data from the StockTick events are a length window and a view for computing statistics on volume. The name for the event stream is "volumeStats".

```
select * from org.esper.example.StockTick(symbol='IBM').win:length(100).stat:uni('volume') as volumeSt
```

6.4.1. Specifying an event type

In the example above the event type was `org.esper.example.StockTick`. The event type is simply the fully qualified Java class name of the class of the event instances that are send into the runtime. The below example shows one way to obtain the fully qualified class name of a given Java class `CarLocEvent`.

```
String carLocEvent = CarLocEvent.class.getName();
String stmt = "from " + carLocEvent + "().win:length(100)"
```

6.4.2. Specifying event filter criteria

Filter criteria follow the same syntax as outlined in the event pattern section on filters, see Section 5.3, “Filter Expressions”. Filter criteria operators are: `=, < , > , >=, <=`. Ranges use the `in` keyword and round `(...)` or square brackets `[]`.

Esper filters out events in an event stream as defined by filter criteria before it sends events to subsequent views. Thus, compared to search conditions in a where-clause, filter criteria remove unneeded events early.

The below example is a filter criteria list that removes events based on category, price and grade.

```
from mypackage.RfidEvent(category="Perishable", price<1.00, grade in [1, 2])
```

6.4.3. Specifying views

Views are used to derive or aggregate data. Views can be staggered onto each other. The section below outlines the views available and plug-in of custom views.

Views can optionally take one or parameters. These parameters can constist of primitive constants such as String, boolean or numeric types. String arrays are also supported as a view parameter type.

Views can optionally take one or parameters. These parameters can constist of primitive constants such as String, boolean or numeric types. String arrays are also supported as a view parameter type.

The below example uses the car location event. It specifies an empty list of filter criteria by adding a empty round brackets `()` after the event type. The first view `"std:group('carId')"` groups car location events by car id. The second view `"win:length(4)"` keeps a length window of the 4 last events, with one length window for each car id. The next view `"std:group({'expressway', 'direction', 'segment'})"` groups each event by it's expressway, direction and segment property values. Again, the grouping is done for each car id considering the last 4 events only. The last view `"std:size()"` is used to report the number of events. Thus the below example reports the number of events per car id and per expressway, direction and segment considering the last 4 events for each car id only. The `"as accSegment"` syntax assigns the name accSegment to the resulting event stream.

```
String carLocEvent = CarLocEvent.class.getName();
String joinStatement = "select * from " + carLocEvent +
            "().std:group('carId').win:length(4).std:group({'expressway', 'direction', 'segment'}).std
```

6.5. Specifying Search Conditions : the *Where* Clause

The where clause is optional in EQL statements. Via the where clause event streams can be joined and events can be filtered.

Currently only comparison operators `=, < , > , >=, <=` and logical combinations via `and` and `or` are supported in the where clause. The where keyword can also introduce join conditions as outlined in Section 6.7, “Joining Event Streams”.

6.6. Build-in views

This chapter outines the views that are built into Esper.

6.6.1. Window views

Length window

Creates a moving window extending the specified number of elements into the past.

The below example calculates basic univariate statistics for the last 5 stock ticks for symbol IBM.

```
StockTickEvent(symbol='IBM').win:length(5).stat:uni('price')
```

Time window

The time_window creates a moving time window extending from the specified time interval in seconds into the past based on the system time.

For the IBM stock tick events in the last 1000 milliseconds, calculate statistics on price.

```
StockTickEvent(symbol='IBM').win:time(1).stat:uni('price')
```

Externally-timed window

Similar to the time window this view moving time window extending from the specified time interval in seconds into the past, but based on the millisecond time value supplied by an event property.

This view holds stock tick events of the last 10 seconds based on the timestamp property in `StockTickEvent`.

```
StockTickEvent().win:ext_timed(10, 'timestamp')
```

Time window buffer

This window view buffers events and releases them every specified time interval in one update.

Batch events into a 5 second window releasing new batches every 5 seconds. Listeners to updates posted by this view receive updated information only every 5 seconds.

```
StockTickEvent().win:time_batch(5)
```

6.6.2. Standard view set

Unique

A view that includes only the most recent among events having the same value for the specified field.

The below example creates a view that retains only the last event per symbol.

```
StockTickEvent().std:unique('symbol')
```

Group

This view groups events into sub-views by the value of the specified field.

This example calculates statistics on price separately for each symbol.

```
StockTickEvent().std:group('symbol').stat:uni('price')
```

Size

This view returns the number of elements in view.

This example view reports the number of events within the last 1 minute.

```
StockTickEvent().win:time(60000).std:size()
```

Last

This view exposes the last element of its parent view.

This example view contains the retains the statistics calculated on stock tick price for the symbol IBM.

```
StockTickEvent(symbol='IBM').stat:uni('price').std:last()
```

6.6.3. Statistics views

Univariate statistics

This view calculated basic univariate statistics on an event property.

Table 6.1. Univariate statistics derived properties

Property Name	Description
`count`	Number of values
`sum`	Sum of values
`average`	Average of values
`variance`	Variance
`stdev`	Sample standard deviation (square root of variance)
`stdevpa`	Population standard deviation

The below example calculates price statistics on stock tick events for the last 10 events.

```
StockTickEvent().win:length(10).stat:uni('price')
```

Regression

This view calculates regression on two event properties.

Table 6.2. Regression derived properties

Property Name	Description
`slope`	Slope
`yintercept`	Y Intercept

Calculate slope and y-intercept on price and offer for all events in the last 10 seconds.

```
StockTickEvent().win:time(10000).stat:linest('price', 'offer')
```

Correlation

Calculates the correlation on two event properties.

Table 6.3. Correlation derived properties

Property Name	Description
`correl`	Correlation between two event properties

Calculate correlation on price and offer over all stock tick events for IBM.

```
StockTickEvent(symbol='IBM').stat:correl('price', 'offer')
```

Weighted average

Returns the weigthed average given a weight field and a field to compute the average for. Syntax: weighted_avg(field, weightField)

Table 6.4. Weighted average derived properties

Property Name	Description
`average`	Weighted average

Views that derive the volume-weighted average price for the last 3 seconds.

```
StockTickEvent(symbol='IBM').win:time(3000).stat:weighted_avg('price', 'volume')
```

Multi-dimensional statistics

This view works similar to the `std:group` views in that it groups information by one or more event properties. The view accepts 3 or more parameters: The first parameter to the view defines the univariate statistics values to derive. The second parameter is the property name to derive data from. The remaining parameters supply the event property names to use to derive dimensions.

Table 6.5. Multi-dim derived properties

Property Name	Description
`cube`	The cube following the interface

The example below derives the count, average and standard deviation latency of service measurement events per customer.

```
ServiceMeasurement().stat:multidim_stats({‘count’, ‘average’, ‘stdev’},
    'latency', 'customer')
```

This example derives the average latency of service measurement events per customer, service and error status for events in the last 30 seconds.

```
ServiceMeasurement().win:lenght(30000).stat:multidim_stats({‘average’},
        'latency', 'customer', 'service', 'status')
```

6.6.4. Extension View Set

Sorted Window View

This view sorts by values in the specified event property and keeps only the top elements up to the given size.

The syntax for this view is : `sort(String propertyName, boolean isDescending, int size) .`

These view can be used to sort on price descending keeping the lowest 10 prices and reporting statistics on price.

```
StockTickEvent().ext:sort(“price”, true, 10).stat:uni(“price”))
```

6.7. Joining Event Streams

As outlined in the EQL syntax earlier, two or more event streams can be part of the from clause and by used to determine the resulting events. The where clause is lists the join conditions that Esper uses to relate events in two or more streams.

The below example is a fairly complex join of 2 streams in which the where clause dictates how to join the accSeg and the curCarSeg event streams.

```
String carLocEvent = CarLocEvent.class.getName();
String joinStatement = "select * from " +
                carLocEvent + "().std:group('carId').win:length(4).std:group({'expressway', 'direction
                carLocEvent + "().win:time(30).std:unique('carId') as curCarSeg" +
                        " where accSeg.size >= 4" +
                        "   and accSeg.expressway = curCarSeg.expressway" +
                        "   and accSeg.direction = curCarSeg.direction" +
                        "   and (" +
                                        "(accSeg.direction=0 " +
                                        " and curCarSeg.segment < accSeg.segment" +
                                        " and curCarSeg.segment > accSeg.segment - 5)" +
                                " or " +
                                        "(accSeg.direction=1 " +
                                        " and curCarSeg.segment > accSeg.segment" +
                                        " and curCarSeg.segment < accSeg.segment + 5)" +
                                ")";
```

6.8. View Plug-in

This is currently not supported (planned).

Chapter 7. Adapters

This chapters discusses the adapters.

7.1. Adapter

Adapters adapt event executions in the outside world into a format for processing by Esper, and feed events to Esper.

Currently there are no pre-build adapters available for Esper.

Chapter 8. Indicators

8.1. Intro

Indicators are pluggable modules that communicate the results of event stream processing to the external world. Indicators can act as visualizers that present a graphical view of their event inputs. They can also be warning agents (monitors) that send alerts, warnings or other control events to the outside world.

In their implementation indicators can be classes that implement the `UpdateListener` interface and that can thus be attached directly to one or more statements. Indicactors can also be attached to one or more `EPStatement` instances. This makes is possible for indicators to merge data as well as pull data from trigger and statement views.

Indicators may be integration components that plug together with other software, an some indicators will be supplied by Esper. Esper currently only has one indicator module as described below.

8.2. JMX Indicator

The `net.esper.indicator.jmx.JMXLastEventIndicator` displays the last event in a JMX MBean it registers with the MBeanServer obtained via ManagementFactory.getPlatformMBeanServer();

Chapter 9. Examples

9.1. Overview

This chapters outlines the examples that come with Esper in the `eg/src` folder of the distribution. The code for examples can be found in the `net.esper.example` packages.

JUnit tests exist for the example code. The JUnit test source code for the examples can be found in the `eg/test` folder. To build and run the examples and their JUnit tests, use the Maven 2 goal `test`. The JUnit test source code can also be helpful in understanding the example and in the use of Esper APIs.

9.2. StockTicker

The StockTicker example comes from the stock trading domain. The example creates event patterns to filter stock tick events based on price and symbol. When a stock tick event is encountered that falls outside the lower or upper price limit, the example simply displays that stock tick event. The price range itself is dynamically created and changed. This is accomplished by an event patterns that searches for another event class, the price limit event.

The classes `net.esper.example.stockticker.event.StockTick` and `PriceLimit` represent our events. The event patterns are created by the class `net.esper.example.stockticker.monitor.StockTickerMonitor`.

Summary:

- Good example to learn the API and get started with event patterns
- Dynamically creates and removes event patterns based on price limit events received
- Simple, highly-performant filter expressions for event properties in the stock tick event such as symbol and price

9.3. MatchMaker

In the MatchMaker example every mobile user has an X and Y location, a set of properties (gender, hair color, age range) and a set of preferences (one for each property) to match. The task of the event patterns created by this example is to detect mobile users that are within proximity given a certain range, and for which the properties match preferences.

The event class representing mobile users is `net.esper.example.matchmaker.event.MobileUserBean`. The `net.esper.example.matchmaker.monitor.MatchMakingMonitor` class contains the patterns for detecing matches.

Summary:

- Dynamically creates and removes event patterns based on mobile user events received
- Uses range matching for X and Y properties of mobile user events

9.4. QualityOfService

This examples develops some code for measuring quality-of-service levels such as for a service-level agree-

ment (SLA). A SLA is a contract between 2 parties that defines service constraints such as maximum latency for service operations or error rates.

The example measures and monitors operation latency and error counts per customer and operation. When one of our operations oversteps these constraints, we want to be alerted right away. Additionally, we would like to have some monitoring in place that checks the health of our service and provides some information on how the operations are used.

Some of the constraints we need to check are:

- That the latency (time to finish) of some of the operations is always less then X seconds.
- That the latency average is always less then Y seconds over Z operation invocations.

The `net.esper.example.qos_sla.events.OperationMeasurement` event class with it's latency and status properties is the main event used for the SLA analysis. The other event `LatencyLimit` serves to set latency limits on the fly.

The `net.esper.example.qos_sla.monitor.AverageLatencyMonitor` creates an EQL statement that computes latency statistics per customer and operation for the last 100 events. The `DynaLatencySpikeMonitor` uses an event pattern to listen to spikes in latency with dynamically set limits.The `ErrorRateMonitor` uses the timer `'at'` operator in an event pattern that wakes up periodically and polls the error rate within the last 10 minutes. The `ServiceHealthMonitor` simply alerts when 3 errors occur, and the `SpikeAndErrorMonitor` alerts when a fixed latency is overstepped or an error status is reported.

Summary:

- This example combines event patterns with EQL statements for event stream analysis.
- Shows the use of the timer `'at'` operator and followed-by operator `->` in event patterns
- Outlines basic EQL statements
- Shows how to pull data out of EQL statements rather then subscribing to events a statement publishes

9.5. LinearRoad

The Linear Road example is a very incomplete implementation of the Stream Data Management Benchmark [3] by Standford University.

Linear Road simulates a toll system for the motor vehicle expressways of a large metropolitan area. The main event in this example is a car location report which the class `net.esper.example.linearroad.CarLocEvent` represents. Currently the event stream joins are performed by JUnit test classes in the `eg/test` folder. See the `net.esper.example.linearroad.TestAccidentNotify` and the `TestCarSegmentCount` classes. Please consider this a work in progress.

Summary:

- Shows more complex joins between event streams.

Chapter 10. References

10.1. Reference List

- Luckham, David. 2002. *The Power of Events.* Addison-Wesley.
- The Stanford Rapide (TM) Project. *http://pavg.stanford.edu/rapide*.
- Arasu, Arvind, et.al.. 2004. Linear Road: A Stream Data Management Benchmark, Stanford University *http://www.cs.brown.edu/research/aurora/Linear_Road_Benchmark_Homepage.html*.