Esper Reference Documentation

Version: 1.10.0

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.

If you are new to Esper, please follow these steps:

1. Read the tutorials, case studies and solution patterns available on the Esper public web site at `http://esper.codehaus.org`
2. 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)
3. Read Section 6.1, “EQL Introduction” for an introduction to event stream processing via EQL
4. Read Section 7.1, “Event Pattern Overview” for an overview over event patterns
5. Read Chapter 4, *Understanding the Output Model* to gain insight into EQL continuous query results
6. Then glance over the examples Section 9.1, “Examples 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 fabrication 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 designed for similarity with the SQL query language but differs from SQL in its 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.

1.4. Required 3rd Party Libraries

Esper requires the following 3rd-party libraries at runtime:

- 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.

Esper requires the following 3rd-party libraries at compile-time and for running the test suite:

- JUnit is a great unit testing framework. Its license has also been placed in the lib directory. The library is required for build-time only.
- MySQL connector library is used for testing SQL integration and is required for running the automated test suite.

Chapter 2. Configuration

Esper engine configuration is entirely optional. Esper has a very small number of configuration parameters that can be used to simplify event pattern and EQL statements, and to tune the engine behavior to specific requirements. The Esper engine works out-of-the-box without configuration.

2.1. Programmatic Configuration

An instance of `net.esper.client.Configuration` represents all configuration parameters. The `Configuration` is used to build an (immutable) `EPServiceProvider`, which provides the administrative and runtime interfaces for an Esper engine instance.

You may obtain a `Configuration` instance by instantiating it directly and adding or setting values on it. The `Configuration` instance is then passed to `EPServiceProviderManager` to obtain a configured Esper engine.

```
Configuration configuration = new Configuration();
configuration.addEventTypeAlias("PriceLimit", PriceLimit.class.getName());
configuration.addEventTypeAlias("StockTick", StockTick.class.getName());
configuration.addImport("org.mycompany.mypackage.MyUtility");
configuration.addImport("org.mycompany.util.*");

EPServiceProvider epService = EPServiceProviderManager.getProvider("sample", configuration);
```

Note that `Configuration` is meant only as an initialization-time object. The Esper engine represented by an `EPServiceProvider` is immutable and does not retain any association back to the `Configuration`.

2.2. Configuration via XML File

An alternative approach to configuration is to specify a configuration in an XML file.

The default name for the XML configuration file is `esper.cfg.xml`. Esper reads this file from the root of the `CLASSPATH` as an application resource via the `configure` method.

```
Configuration configuration = new Configuration();
configuration.configure();
```

The `Configuration` class can read the XML configuration file from other sources as well. The `configure` method accepts `URL, File and String` filename parameters.

```
Configuration configuration = new Configuration();
configuration.configure("myengine.esper.cfg.xml");
```

2.3. XML Configuration File

Here is an example configuration file. The schema for the configuration file can be found in the `etc` folder and is named `esper-configuration-1-0`.

```
<?xml version="1.0" encoding="UTF-8"?>
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="esper-configuration-1-0.xsd">
  <event-type alias="StockTick" class="net.esper.example.stockticker.event.StockTick"/>
  <event-type alias="PriceLimit" class="net.esper.example.stockticker.event.PriceLimit"/>
  <auto-import import-name="org.mycompany.mypackage.MyUtility"/>
```

```
  <auto-import import-name="org.mycompany.util.*"/>
</esper-configuration>
```

The example above is only a subset of the configuration items available. The next chapters outline the available configuration in greater detail.

2.4. Configuration Items

2.4.1. Events represented by Java Classes

Event type alias to Java class mapping

This configuration item can be used to allow event pattern statements and EQL statements to use an event type alias rather then the fully qualified Java class name. Note that Java Interface classes and abstract classes are also supported as event types via the fully qualified Java class name, and an event type alias can also be defined for such classes.

The example pattern statement below first shows a pattern that uses the alias `StockTick`. The second pattern statement is equivalent but specifies the fully-qualified Java class name.

```
every StockTick(symbol='IBM')"
```

```
every net.esper.example.stockticker.event.StockTick(symbol='IBM')
```

The event type alias can be listed in the XML configuration file as shown below. The `Configuration` API can also be used to programatically specify an event type alias, as shown in an earlier code snippet.

```
<event-type alias="StockTick" class="net.esper.example.stockticker.event.StockTick"/>
```

Non-JavaBean and Legacy Java Event Classes

Esper can process Java classes that provide event properties through other means then through JavaBean-style getter methods. It is not necessary that the method and member variable names in your Java class adhere to the JavaBean convention - any public methods and public member variables can be exposed as event properties via the below configuration.

A Java class can optionally be configured with an accessor style attribute. This attribute instructs the engine how it should expose methods and fields for use as event properties in statements.

Table 2.1. Accessor Styles

Style Name	Description
`javabean`	As the default setting, the engine exposes an event property for each public method following the JavaBean getter-method conventions
`public`	The engine exposes an event property for each public method and public member variable of the given class
`explicit`	The engine exposes an event property only for the explicitly configured public methods and public member variables

Using the `public` setting for the `accessor-style` attribute instructs the engine to expose an event property for each public method and public member variable of a Java class. The engine assigns event property names of the same name as the name of the method or member variable in the Java class.

For example, assuming the class `MyLegacyEvent` exposes a method named `readValue` and a member variable named `myField`, we can then use properties as shown.

```
select readValue, myField from MyLegacyEvent
```

Using the `explicit` setting for the `accessor-style` attribute requires that event properties are declared via configuration. This is outlined in the next chapter.

When configuring an engine instance from an XML configuration file, the XML snippet below demonstrates the use of the `legacy-type` element and the `accessor-style` attribute.

```
<event-type alias="MyLegacyEvent" class="com.mycompany.mypackage.MyLegacyEventClass">
  <legacy-type accessor-style="public"/>
</event-type>
```

When configuring an engine instance via Configuration API, the sample code below shows how to set the accessor style.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.setAccessorStyle(ConfigurationEventTypeLegacy.AccessorStyle.PUBLIC);
config.addEventTypeAlias("MyLegacyEvent", MyLegacyEventClass.class.getName(), legacyDef);

EPServiceProvider epService = EPServiceProviderManager.getProvider("sample", configuration);
```

Specifying Event Properties for Java Classes

Sometimes it may be convenient to use event property names in pattern and EQL statements that are backed up by a given public method or member variable (field) in a Java class. And it can be useful to declare multiple event properties that each map to the same method or member variable.

We can configure properties of events via `method-property` and `field-property` elements, as the next example shows.

```
<event-type alias="StockTick" class="net.esper.example.stockticker.event.StockTickEvent">
        <legacy-type accessor-style="javabean" code-generation="enabled">
                <method-property name="price" accessor-method="getCurrentPrice" />
                <field-property name="volume" accessor-field="volumeField" />
        </legacy-type>
</event-type>
```

The XML configuration snippet above declared an event property named `price` backed by a getter-method named `getCurrentPrice`, and a second event property named `volume` that is backed by a public member variable named `volumeField`. Thus the price and volume properties can be used in a statement:

```
select avg(price * volume) from StockTick
```

As with all configuration options, the API can also be used:

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.addMethodProperty("price", "getCurrentPrice");
legacyDef.addFieldProperty("volume", "volumeField");
config.addEventTypeAlias("StockTick", StockTickEvent.class.getName(), legacyDef);
```

Turning off Code Generation

Esper employes the `CGLIB` library for very fast read access to event property values. For certain legacy Java classes it may be desirable to disable the use of this library and instead use Java reflection to obtain event property values from event objects.

In the XML configuration, the optional `code-generation` attribute in the `legacy-type` section can be set to `disabled` as shown next.

```
<event-type alias="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass">
        <legacy-type accessor-style="javabean" code-generation="disabled" />
</event-type>
```

The sample below shows how to configure this option via the API.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.setCodeGeneration(ConfigurationEventTypeLegacy.CodeGeneration.DISABLED);
config.addEventTypeAlias("MyLegacyEvent", MyLegacyEventClass.class.getName(), legacyDef);
```

Case Sensitivity and Property Names

By default the engine resolves Java event properties case sensitive. That is, property names in statements must match JavaBean-convention property names in name and case. This option controls case sensitivity per Java class.

In the configuration XML, the optional `property-resolution-style` attribute in the `legacy-type` element can be set to any of these values:

Table 2.2. Property Resolution Case Sensitivity Styles

Style Name	Description
`case_sensitive (default)`	As the default setting, the engine matches property names for the exact name and case only.
`case_insensitive`	Properties are matched if the names are identical. A case insensitive search is used and will choose the first property that matches the name exactly or the first property that matches case insensitively should no match be found.
`distinct_case_insensitive`	Properties are matched if the names are identical. A case insensitive search is used and will choose the first property that matches the name exactly case insensitively. If more than one 'name' can be mapped to the property an exception is thrown.

The sample below shows this option in XML configuration, however the setting can also be changed via API:

```
<event-type alias="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass">
  <legacy-type property-resolution-style="case_insensitive"/>
</event-type>
```

2.4.2. Events represented by `java.util.Map`

The engine can process `java.util.Map` events via the `sendEvent(Map map, String eventTypeAlias)` method on the `EPRuntime` interface. Entries in the Map represent event properties. Keys must be of type `java.util.String` for the engine to be able to look up event property names in pattern or EQL statements. Values can be of any type. JavaBean-style objects as values in a `Map` can also be processed by the engine. Please see the Chapter 5, *Event Representations* section for details on how to use `Map` events with the engine.

Via configuration we provide an event type alias name for `Map` events for use in statements, and the event property names and types enabling the engine to validate properties in statements.

The below snippet of XML configuration configures an event named `MyMapEvent`.

```
<event-type alias="MyMapEvent">
  <java-util-map>
    <map-property name="carId" class="int"/>
    <map-property name="carType" class="string"/>
    <map-property name="assembly" class="com.mycompany.Assembly"/>
  </java-util-map>
</event-type>
```

This configuration defines the `carId` property of `MyMapEvent` events to be of type `int`, and the `carType` property to be of type `java.util.String`. The `assembly` property of the Map event will contain instances of `com.mycompany.Assembly` for the engine to query.

The valid list of values for the type definition via the `class` attribute is:

- `string` or `java.lang.String`
- `char` or `java.lang.Character`
- `byte` or `java.lang.Byte`
- `short` or `java.lang.Short`
- `int` or `java.lang.Integer`
- `long` or `java.lang.Long`
- `float` or `java.lang.Float`
- `double` or `java.lang.Double`
- `boolean` or `java.lang.Boolean`
- Any fully-qualified Java class name that can be resolved by the engine via `Class.forName`

You can also use the configuration API to configure `Map` event types, as the short code snippet below demonstrates.

```
Properties properties = new Properties();
properties.put("carId", "int");
properties.put("carType", "string");
properties.put("assembly", Assembly.class.getName());

Configuration configuration = new Configuration();
configuration.addEventTypeAlias("MyMapEvent", properties);
```

Finally, here is a sample EQL statement that uses the configured `MyMapEvent` map event. This statement uses the `chassisTag` and `numParts` properties of `Assembly` objects in each map.

```
select carType, assembly.chassisTag, count(assembly.numParts) from MyMapEvent.win:time(60 sec)
```

2.4.3. Events represented by `org.w3c.dom.Node`

Via this configuration item the Esper engine can natively process `org.w3c.dom.Node` instances, i.e. XML document object model (DOM) nodes. Please see the Chapter 5, *Event Representations* section for details on how to

use `Node` events with the engine.

Esper allows configuring XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name by which their result values will be available for use in expressions.

For XML documents that follow an XML schema, Esper can load and interrogate your schema and validate event property names and types against the schema information.

Nested, mapped and indexed event properties are also supported in expressions against `org.w3c.dom.Node` events. Thus XML trees can conveniently be interrogated using the existing event property syntax for querying JavaBean objects, JavaBean object graphs or `java.util.Map` events.

In the simplest form, the Esper engine only requires a configuration entry containing the root element name and the event type alias in order to process `org.w3c.dom.Node` events:

```
<event-type alias="MyXMLNodeEvent">
  <xml-dom root-element-name="myevent" />
</event-type>
```

You can also use the configuration API to configure XML event types, as the short example below demonstrates. In fact, all configuration options available through XML configuration can also be provided via setter methods on the `ConfigurationEventTypeXMLDOM` class.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setRootElementName("myevent");
desc.addXPathProperty("name1", "/element/@attribute", XPathConstants.STRING);
desc.addXPathProperty("name2", "/element/subelement", XPathConstants.NUMBER);
configuration.addEventTypeAlias("MyXMLNodeEvent", desc);
```

The next example presents all relevant configuration options in a sample configuration entry.

```
<event-type alias="AutoIdRFIDEvent">
  <xml-dom root-element-name="Sensor" schema-resource="data/AutoIdPmlCore.xsd"
       default-namespace="urn:autoid:specification:interchange:PMLCore:xml:schema:1">
    <namespace-prefix prefix="pmlcore"
       namespace="urn:autoid:specification:interchange:PMLCore:xml:schema:1"/>
    <xpath-property property-name="countTags"
       xpath="count(/pmlcore:Sensor/pmlcore:Observation/pmlcore:Tag)" type="number"/>
  </xml-dom>
</event-type>
```

This example configures an event property named `countTags` whose value is computed by an XPath expression. The namespace prefixes and default namespace are for use with XPath expressions and must also be made known to the engine in order for the engine to compile XPath expressions. Via the `schema-resource` attribute we instruct the engine to load a schema file.

Here is an example EQL statement using the configured event type named `AutoIdRFIDEvent`.

```
select ID, countTags from AutoIdRFIDEvent.win:time(30 sec)
```

Schema Resource

The `schema-resource` attribute takes a schema resource URL or classpath-relative filename. The engine attempts to resolve the schema resource as an URL. If the schema resource name is not a valid URL, the engine attempts to resolve the resource from classpath via the `ClassLoader.getResource` method using the thread context class loader. If the name could not be resolved, the engine uses the Configuration class classloader.

By configuring a schema file for the engine to load, the engine performs these additional services:

- Validates the event properties in a statement, ensuring the event property name matches an attribute or element in the XML
- Determines the type of the event property allowing event properties to be used in type-sensitive expressions such as expressions involving arithmatic (Note: XPath properties are also typed)
- Matches event property names to either element names or attributes

If no schema resource is specified, none of the event properties specified in statements are validated at statement creation time and their type defaults to `java.lang.String`. Also, attributes are not supported if no schema resource is specified and must thus be declared via XPath expression.

XPath Property

The `xpath-property` element adds event properties to the event type that are computed via an XPath expression. In order for the XPath expression to compile, be sure to specify the `default-namespace` attribute and use the `namespace-prefix` to declare namespace prefixes.

XPath expression properties are strongly typed. The `type` attribute allows the following values. These values correspond to those declared by `javax.xml.xpath.XPathConstants`.

- number (Note: resolves to a `double`)
- string
- boolean

2.4.4. Class and package imports

Esper allows invocations of static Java library functions as outlined in Section 6.14, “Single-row Function Reference”. This configuration item can be set to allow a partial rather than a fully qualified class name in such invocations. The imports work in the same way as in Java files, so both packages and classes can be imported.

```
select Math.max(priceOne, PriceTwo)
// via configuration equivalent to
select java.lang.Math.max(priceOne, priceTwo)
```

Esper auto-imports the following Java library packages if no other configuration is supplied. This list is replaced with any configuration specified in a configuration file or through the API.

- java.lang.*
- java.math.*
- java.text.*
- java.util.*

In an XML configuration file the auto-import configuration may look as below. Note that all configuration options are available through the `Configuration` API as well.

```
<auto-import import-name="com.mycompany.mypackage.*"/>
  <auto-import import-name="com.mycompany.myapp.MyUtilityClass"/>
```

2.4.5. Relational Database Access

Esper has the capability to join event streams against historical data sources, such as a relational database. This section describes the configuration entries that the engine requires to access data stored in your database. Please

see Section 6.13, “Joining Relational Data via SQL” for information on the use of EQL queries that include historical data sources.

EQL queries that poll data from a relational database specify the name of the database as part of the EQL statement. The engine uses the configuration information described here to resolve the database name in the statement to database settings. The required and optional database settings are summarized below.

- Database connections can be obtained via JDBC `javax.xml.DataSource` or alternatively via `java.sql.DriverManager`. Either one of these methods to obtain new database connections is a required configuration.
- Optionally, JDBC connection-level settings such as auto-commit, transaction isolation level, read-only and the catalog name can be defined.
- Optionally, a connection lifecycle can be set to indicate to the engine whether the engine must retain connections or must obtain a new connection for each lookup.
- Optionally, define a cache policy to allow the engine to retrieve data from a query cache, reducing the number of query executions.

Some of the settings can have important performance implications that need to be carefully considered in relationship to your database software, JDBC driver and runtime environment. This section attempts to outline such implications where appropriate.

The sample XML configuration file in the "etc" folder can be used as a template for configuring database settings. All settings are also available by means of the configuration API through the classes `Configuration` and `ConfigurationDBRef`.

Connections obtained via DataSource

The snippet of XML below configures a database named `mydb1` to obtain connections via a `javax.sql.DataSource`. The `datasource-connection` element instructs the engine to obtain new connections to the database `mydb1` by performing a lookup via `javax.naming.InitialContext` for the given object lookup name. Optional environment properties for the `InitialContext` are also shown in the example.

```
<database-reference name="mydb1">
  <datasource-connection context-lookup-name="java:comp/env/jdbc/mydb">
    <env-property name="java.naming.factory.initial" value ="com.myclass.CtxFactory"/>
    <env-property name="java.naming.provider.url" value ="iiop://localhost:1050"/>
  </datasource-connection>
</database-reference>
```

To help you better understand how the engine uses this information to obtain connections, we have included the logic below.

```
if (envProperties.size() > 0) {
  initialContext = new InitialContext(envProperties);
}
else {
  initialContext = new InitialContext();
}
DataSource dataSource = (DataSource) initialContext.lookup(lookupName);
Connection connection = dataSource.getConnection();
```

Connections obtained via DriverManager

The next snippet of XML configures a database named `mydb2` to obtain connections via `java.sql.DriverManager`. The `drivermanager-connection` element instructs the engine to obtain new connections to the database `mydb2` by means of `Class.forName` and `DriverManager.getConnection` using the class

name, URL and optional username, password and connection arguments.

```
<database-reference name="mydb2">
  <drivermanager-connection class-name="my.sql.Driver"
        url="jdbc:mysql://localhost/test?user=root&amp;password=mypassword"
        user="myuser" password="mypassword">
    <connection-arg name="user" value ="myuser"/>
    <connection-arg name="password" value ="mypassword"/>
    <connection-arg name="somearg" value ="someargvalue"/>
  </drivermanager-connection>
</database-reference>
```

The username and password are shown in multiple places in the XML only as an example. Please check with your database software on the required information in URL and connection arguments.

Connections-level settings

Additional connection-level settings can optionally be provided to the engine which the engine will apply to new connections. When the engine obtains a new connection, it applies only those settings to the connection that are explicitly configured. The engine leaves all other connection settings at default values.

The below XML is a sample of all available configuration settings. Please refer to the Java API JavaDocs for `java.sql.Connection` for more information to each option or check the documentation of your JDBC driver and database software.

```
<database-reference name="mydb2">
... configure data source or driver manager settings...
  <connection-settings auto-commit="true" catalog="mycatalog"
      read-only="true" transaction-isolation="1" />
</database-reference>
```

The `read-only` setting can be used to indicate to your database engine that SQL statements are read-only. The `transaction-isolation` and `auto-commit` help you database software perform the right level of locking and lock release. Consider setting these values to reduce transactional overhead in your database queries.

Connections lifecycle settings

By default the engine retains a separate database connection for each started EQL statement. However, it is possible to override this behavior and require the engine to obtain a new database connection for each lookup, and to close that database connection after the lookup is completed. This often makes sense when you have a large number of EQL statements and require pooling of connections via a connection pool. If your runtime environment includes an application server, the connection pool may be exposed as a `DataSource`.

The XML for this option is below. The connection lifecycle allows the following values: `pooled` and `retain`.

```
<database-reference name="mydb2">
... configure data source or driver manager settings...
    <connection-lifecycle value="pooled"/>
</database-reference>
```

Cache settings

Cache settings can dramatically reduce the number of database queries that the engine executes for EQL statements. If no cache setting is specified, the engine does not cache query results and executes a separate database query for every event.

Caches store the results of database queries and make these results available to subsequent queries using the exact same query parameters as the query for which the result was stored. If your query returns one or more rows,

the cache keep the result rows of the query keyed to the parameters of the query. If your query returns no rows, the cache also keeps the empty result. Query results are held by a cache until the cache entry is evicted. The strategies available for evicting cached query results are listed next.

LRU Cache

The least-recently-used (LRU) cache is configured by a maximum size. The cache discards the least recently used query results first once the cache reaches the maximum size.

The XML configuration entry for a LRU cache is as below. This entry configures an LRU cache holding up to 1000 query results.

```
<database-reference name="mydb">
... configure data source or driver manager settings...
    <lru-cache size="1000"/>
</database-reference>
```

Expiry-time Cache

The expiry time cache is configured by a maximum age in seconds and a purge interval. The cache discards (on the get operation) any query results that are older then the maximum age so that stale data is not used. If the cache is not empty, then every purge interval number of seconds the engine purges any expired entries from the cache.

The XML configuration entry for an expiry-time cache is as follows. The example configures an expiry time cache in which prior query results are valid for 60 seconds and which the engine inspects every 2 minutes to remove query results older then 60 seconds.

```
<database-reference name="mydb">
... configure data source or driver manager settings...
    <expiry-time-cache max-age-seconds="60" purge-interval-seconds="120"/>
</database-reference>
```

2.4.6. Engine Settings related to Concurrency and Threading

Preserving the order of events delivered to listeners

In multithreaded environments, this setting controls whether dispatches of statement result events to listeners preserve the ordering in which a statement processes events. By default the engine guarantees that it delivers a statement's result events to statement listeners in the order in which the result is generated. This behavior can be turned off via configuration as below.

The next code snippet shows how to control this feature:

```
Configuration config = new Configuration();
config.getEngineDefaults().getThreading().setListenerDispatchPreserveOrder(false);
engine = EPServiceProviderManager.getDefaultProvider(config);
```

An the XML configuration file can also control this feature by adding the following elements:

```
<engine-settings>
  <defaults>
    <threading>
      <listener-dispatch preserve-order="false" timeout-msec="2000"/>
      <insert-into-dispatch preserve-order="false"/>
    </threading>
  </defaults>
```

```
</engine-settings>
```

As discussed, by default the engine can temporarily block a thread when delivering result events to listeners in order to preserve the order in which results are generated by a given statement. The maximum time the engine blocks a thread can also be configured, and by default is set to 1 second.

Preserving the order of events for insert-into streams

In multithreaded environments, this setting controls whether insert-into streams preserve the order of events inserted into them by one or more statements, allowing statements that consume other statement's events to behave deterministic.

By default, the engine acquires a lock per insert-into stream when a statement makes events available to further statements using the `insert into` clause. The lock allows generated events to be processed by further statements consuming the insert-into stream in the order the generating statement(s) produce events. This allows statements that require order (such as pattern detection, previous and prior functions) to behave deterministically.

The setting can be changed via the configuration API and XML as shown in the prior section.

Internal Timer Settings

This option can be used to disable the internal timer thread and such have the application supply external time events, as well as to set a timer resolution.

The next code snippet shows how to disable the internal timer thread via the configuration API:

```
Configuration config = new Configuration();
  config.getEngineDefaults().getThreading().setInternalTimerEnabled(false);
```

This snippet of XML configuration leaves the internal timer enabled (the default) and sets a resolution of 200 milliseconds (the default is 100 milliseconds):

```
<engine-settings>
  <defaults>
    <threading>
      <internal-timer enabled="true" msec-resolution="200"/>
    </threading>
  </defaults>
</engine-settings>
```

2.4.7. Engine Settings related to Event Metadata

Java Class Property Names and Case Sensitivity

As discussed in Section 2.4.1.5, “Case Sensitivity and Property Names” this setting controls case sensitivity for Java event class properties of all Java classes as a default, rather then at a class level.

The next code snippet shows how to control this feature via the API:

```
Configuration config = new Configuration();
config.getEngineDefaults().getEventMeta().setClassPropertyResolutionStyle(
    Configuration.PropertyResolutionStyle.CASE_INSENSITIVE);
```

2.4.8. Engine Settings related to View Resources

Sharing View Resources between Statements

The engine by default attempts to optimize resource usage and thus re-uses or shares views between statements that declare same views. However, in multi-threaded environments, this can lead to reduced concurrency as locking for shared view resources must take place. Via this setting this behavior can be turned off for higher concurrency in multi-threaded processing.

The next code snippet outlines the API to turn off view resource sharing between statements:

```
Configuration config = new Configuration();
config.getEngineDefaults().getViewResources().setShareViews(false);
```

2.4.9. Engine Settings related to Logging

Execution Path Debug Logging

By default, the engine does not produce debug output for the event processing execution paths even when Log4j or Logger configurations have been set to output debug level logs. To enable debug level logging, set this option in the configuration as well as in your Log4j configuration file.

The API to use to enable debug logging is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableExecutionDebug(true);
```

Note: this is a configuration option that applies to all engine instances of a given Java module or VM.

Chapter 3. API Reference

3.1. API Overview

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

Use Esper's administrative interface to create and manage EQL and pattern statements, and set runtime configurations, as discussed in Section 6.1, “EQL Introduction” and Section 7.1, “Event Pattern Overview”.

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.

3.2. Engine Instances

Each instance of an Esper engine is completely independent of other engine instances and has its 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, it creates a new engine instance.

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");
```

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

3.3. The Administrative Interface

3.3.1. Creating Statements

Create event pattern expression and 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 countStmt = admin.createEQL(
  "select count(*) from MarketDataBean.win:time(60 sec)");
```

Note that event pattern expressions can also occur within EQL statements. This is outlined in more detail in Section 6.4.2, “Pattern-based event streams”.

The `create` methods on `EPAdministrator` are overloaded and allow an optional statement name to be passed to the engine. A statement name can be useful for retrieving a statement by name from the engine at a later time. The engine assigns a statement name if no statement name is supplied on statement creation.

The `createPattern` and `createEQL` methods return `EPStatement` instances. Statements 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.

```
countStmt.stop();
countStmt.start();
```

3.3.2. Adding Listeners

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

```
UpdateListener myListener = new MyUpdateListener();
countStmt.addListener(myListener);
```

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

A second listener interface is the `StatementAwareUpdateListener` interface. A `StatementAwareUpdateListener` is especially useful for registering the same listener object with multiple statements, as the listener receives the statement instance and engine instance in addition to new and old data when the engine indicates new results to a listener.

```
StatementAwareUpdateListener myListener = new MyStmtAwareUpdateListener();
statement.addListener(myListener);
```

To indicate results the engine invokes this method on `StatementAwareUpdateListener` listeners: `update(EventBean[] newEvents, EventBean[] oldEvents, EPStatement statement, EPServiceProvider epServiceProvider)`

3.3.3. Using Iterators

Subscribing to events posted by a statement is following a push model. The engine pushes data to listeners 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, an event pattern that fires every 5 seconds could be used to pull data from an EQL statement. The code snippet below demonstrates some pull code.

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

This is a second example:

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

The `iterator` method can be used to pull results out of most statements, including statements that contain aggregation functions, pattern statements, and statements that contain a `where` clause, `group by` clause, `having` clause or `order by` clause.

For statements without an `order by` clause, the `iterator` method returns events in the order maintained by the data window. For statements that contain an `order by` clause, the `iterator` method returns events in the order indicated by the `order by` clause.

Esper places the following restrictions on the pull API and usage of the `iterator` method:

1. EQL statements joining multiple event streams do not support the pull API.
2. Since the `iterator` method returns events to the application immediately, the iterator does not honor an output rate limiting clause, if present.
3. In multithreaded applications, the `iterator` method does not hold any locks and modifications to the underlying data window may throw runtime exceptions in the face of concurrent modifications.

3.3.4. Managing Statements

The `EPAdministrator` interface provides the facilities for managing statements:

- Use the `getStatement` method to obtain an existing started or stopped statement by name
- Use the `getStatementNames` methods to obtain a list of started and stopped statement names
- Use the `startAllStatements`, `stopAllStatements` and `destroyAllStatements` methods to manage all statements in one operation

3.3.5. Runtime Engine Configuration

Certain configuration changes are available to perform on an engine instance while in operation. Such configuration operations are available via the `getConfiguration` method on `EPAdministrator`, which returns an `ConfigurationOperations` object.

The configuration operations available on a running engine instance are as follows. Please see Chapter 2, *Configuration* for more information.

- Add an new event type for a JavaBean class, legacy Java class or custom Java class
- Add an new DOM XML event type
- Add an new Map-based event type

3.4. 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 snippet shows how to send a Java object event to the engine. Note that the `sendEvent` method is overloaded. As events can take on different representation classes in Java, the `sendEvent` takes parameters to reflect the different types of events that can be send into the engine. The Chapter 5, *Event Representations* section explains the types of events accepted.

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

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

Events, in theoretical terms, are observations of a state change that occured in the past. Since one cannot change an event that happened in the past, events are best modelled as immutable objects.

Note that the Esper engine relies on events that are sent into an engine to not change their state. Typically, applications create a new event object for every new event, to represent that new event. Application should not modify an existing event that was sent into the engine.

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. This mechanism is available as a service to enable channel-based publish-subscribe of events emitted from an engine instance via the `emit` method. Emitting events is not integrated with EQL and is available only via the `EPRuntime` interface. Events are emitted on an event channel identified by a name. Listeners are implementations of the `EmittedListener` interface. Via the `addEmittedListener` method a listener can be added to the specified event channel. The lister receives only those events posted to that channel. The channel parameter to `addEmittedListener` also allows null values. If a null channel value is specified, the listeners receives emitted events posted on any channel.

3.5. 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));
```

3.6. 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 its runtime interface.

The `getEventType` method 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 is the events posted for a statement to calculate univariate statistics on an event property. The below example shows such a statement and queries the generated events for an average value.

```
// Derive univariate statistics on price for the last 100 market data events
String stmt = "select * from MarketDataBean(symbol='IBM').win:length(100).stat:uni('price')";
EPStatement priceStatsView = epService.getEPAdministrator().createEQL(stmt);
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 BEvent follows AEvent
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());
        }
}
```

3.7. Engine Threading and Concurrency

Esper is designed from the ground up to operate as a component to multi-threaded, highly-concurrent applications that require efficient use of Java VM resources. In addition, multi-threaded execution requires guarantees in predictability of results and deterministic processing. This section discusses these concerns in detail.

In Esper, an engine instance is a unit of separation. Applications can obtain and discard (initialize) one or more engine instances within the same Java VM and can provide the same or different engine configurations to each instance. An engine instance efficiently shares resources between statements. For example, consider two statements that declare the same data window. The engine matches up view declarations provided by each statements and can thus provide a single data window representation shared between the two statements.

Applications can use Esper APIs to concurrently, by multiple threads of execution, perform such functions as creating and managing statements, or sending events into an engine instance for processing. Applications can use one or more thread pools or any set of same or different threads of execution with any of the public Esper APIs. There are no restrictions towards threading other then those noted in specific sections of this document.

Applications using Esper retain full control over threading, allowing an engine to be easily embedded and used as a component or library in your favorite Java container or process. It is up to the application code to use multiple threads for processing events by the engine, if so desired. All event processing takes places within your application thread call stack. The exception is timer-based processing if your engine instance relies on the internal timer (default).

The fact that event processing takes places within an application thread call stack makes developing applications with Esper easier: Any common Java integrated development environment (IDE) can host an Esper engine instance. This allows developers to easily set up test cases, debug through listener code and inspect input or output events, or trace their call stack.

To send events into an engine concurrently by multiple execution threads, typically applications use the Java `java.lang.Thread` or `java.lang.Runnable` classes or Java 5 concurrent utilities that include abstractions for thread pools and blocking in-memory queues.

Each engine instance maintains a single timer thread (internal timer) providing for time or schedule-based processing within the engine. The default resolution at which the timer operates is 100 milliseconds. The internal timer thread can be disabled and applications can instead send external time events to an engine instance to perform timer or scheduled processing at the resolution required by an application.

Each engine instance performs minimal locking to enable high levels of concurrency. An engine instance locks on a statement level to protect statement resources.

For an engine instance to produce predictable results from the viewpoint of listeners to statements, an engine instance by default ensures that it dispatches statement result events to listeners in the order in which a statement produced result events. Applications that require the highest possible concurrency and do not require predictable order of delivery of events to listeners, this feature can be turned off via configuration.

In multithreaded environments, when one or more statements make result events available via the `insert into` clause to further statements, the engine preserves the order of events inserted into the generated insert-into stream, allowing statements that consume other statement's events to behave deterministic. This feature can also be turned off via configuration.

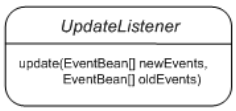
We generally recommended that listener implementations do not block. By implementing listener code as non-blocking code execution threads can often achieve higher levels of concurrency.

Chapter 4. Understanding the Output Model

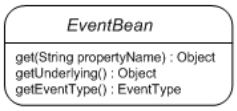
4.1. Introduction

The Esper output model is continuous: Update listeners to statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, views, filters and output rates.

As outlined in Chapter 3, *API Reference* the interface for listeners is `net.esper.client.UpdateListener`. Implementations must provide a single `update` method that the engine invokes when results become available:



The engine provides statement results to update listeners by placing results in `net.esper.event.EventBean` instances. A typical listener implementation queries the `EventBean` instances via getter methods to obtain the statement-generated results.



The `get` method on the `EventBean` interface can be used to retrieve result columns by name. The property name supplied to the `get` method can also be used to query nested, indexed or array properties of object graphs as discussed in more detail in Chapter 5, *Event Representations*.

The `getUnderlying` method on the `EventBean` interface allows update listeners to obtain the underlying event object. For wildcard selects, the underlying event is the event object that was sent into the engine via the `sendEvent` method. For joins and select clauses with expressions, the underlying object implements `java.util.Map`.

4.2. Insert Stream

In this section we look at the output of a very simple EQL statement. The statement selects an event stream without using a data window and without applying any filtering, as follows:

```
select * from Withdrawal
```

This statement selects all `Withdrawal` events. Every time the engine processes an event of type `Withdrawal` or any sub-type of `Withdrawal`, it invokes all update listeners, handing the new event to each of the statement's listeners.

The term *insert stream* denotes the new events arriving, and entering a data window or aggregation. The insert stream in this example is the stream of arriving Withdrawal events, and is posted to listeners as new events.

The diagram below shows a series of Withdrawal events 1 to 6 arriving over time. The number in parenthesis is the withdrawal amount, an event property that is used in the examples that discuss filtering.

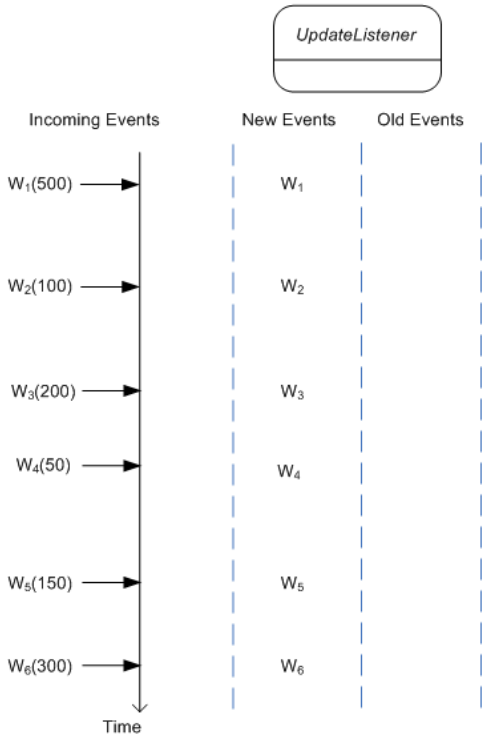


Figure 4.1. Output example for a simple statement

The example statement above results in only new events and no old events posted by the engine to the statement's listeners.

4.3. Insert and Remove Stream

A length window instructs the engine to only keep the last N events for a stream. The next statement applies a length window onto the Withdrawal event stream. The statement serves to illustrate the concept of data window and events entering and leaving a data window:

```
select * from Withdrawal.win:length(5)
```

The size of this statement's length window is five events. The engine enters all arriving Withdrawal events into the length window. When the length window is full, the oldest Withdrawal event is pushed out the window. The engine indicates to listeners all events entering the window as new events, and all events leaving the window as old events.

While the term *insert stream* denotes new events arriving, the term *remove stream* denotes events leaving a data window, or changing aggregation values. In this example, the remove stream is the stream of Withdrawal events that leave the length window, and such events are posted to listeners as old events.

The next diagram illustrates how the length window contents change as events arrive and shows the events posted to an update listener.

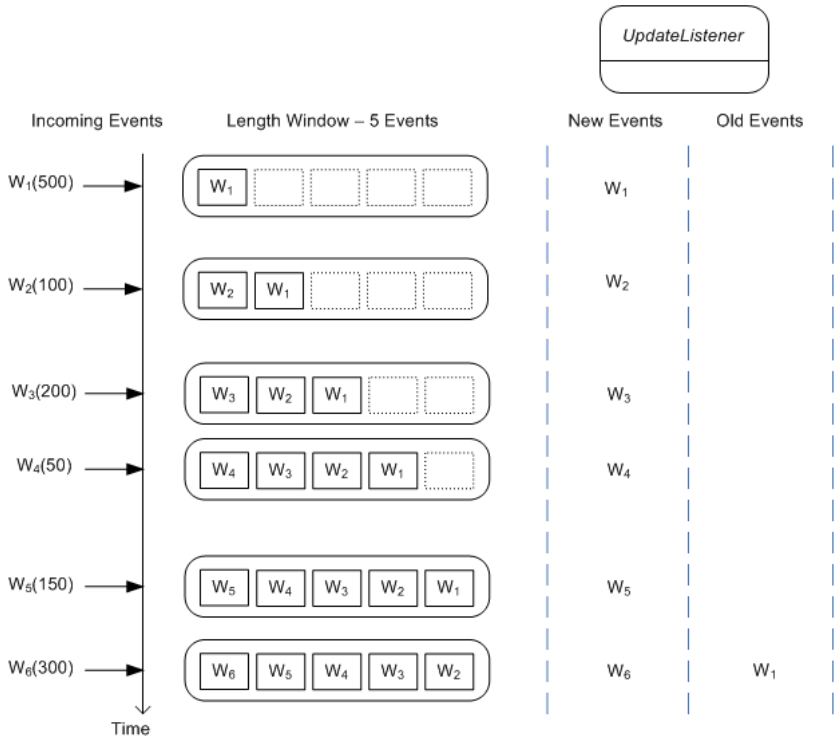


Figure 4.2. Output example for a length window

As before, all arriving events are posted as new events to listeners. In addition, when event W_1 leaves the length window on arrival of event W_6, it is posted as an old event to listeners.

Similar to a length window, a time window also keeps the most recent events up to a given time period. A time window of 5 seconds, for example, keeps the last 5 seconds of events. As seconds pass, the time window actively pushes the oldest events out of the window resulting in one or more old events posted to update listeners.

Note EQL supports optional `istream` and `rstream` keywords on select-clauses and on insert-into clauses. These instruct the engine to only forward events that enter or leave data windows, or select only current or prior aggregation values, i.e. the insert stream or the remove stream.

4.4. Filters and Where-clauses

Filters to event streams allow filtering events out of a given stream before events enter a data window. The statement below shows a filter that selects Withdrawal events with an amount value of 200 or more.

```
select * from Withdrawal(amount>=200).win:length(5)
```

With the filter, any Withdrawal events that have an amount of less then 200 do not enter the length window and are therefore not passed to update listeners. Filters are discussed in more detail in Section 6.4.1, “Filter-based event streams” and Section 7.3, “Pattern Filter Expressions”.

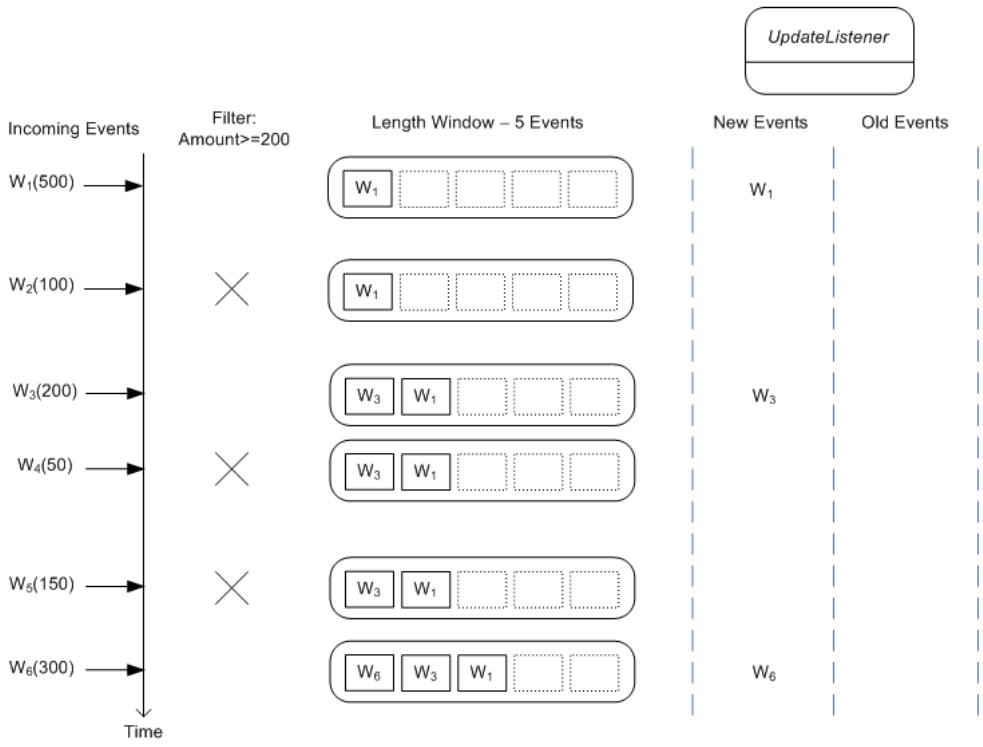


Figure 4.3. Output example for a statement with an event stream filter

The where-clause and having-clause in statements eliminate potential result rows at a later stage in processing, after events have been processed into a statement's data window or other views.

The next statement applies a where-clause to Withdrawal events. Where-clauses are discussed in more detail in Section 6.5, “Specifying Search Conditions: the Where Clause”.

```
select * from Withdrawal.win:length(5) where amount >= 200
```

The where-clause applies to both new events and old events. As the diagram below shows, arriving events enter the window however only events that pass the where-clause are handed to update listeners. Also, as events leave the data window, only those events that pass the conditions in the where-clause are posted to listeners as old events.

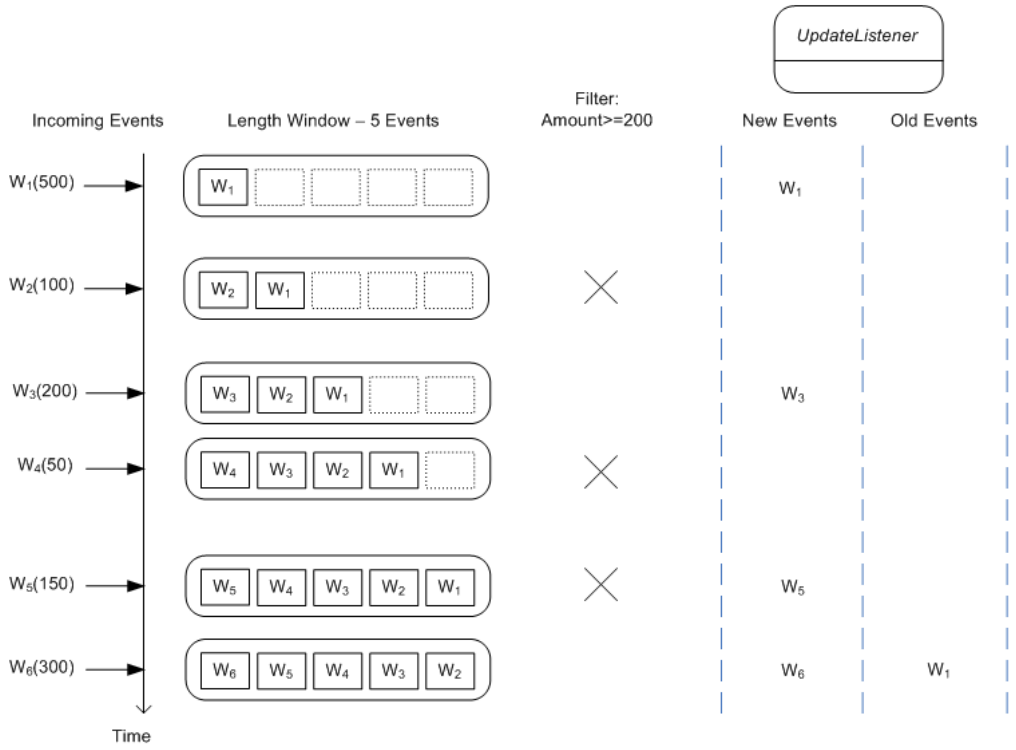


Figure 4.4. Output example for a statement with where-clause

The where-clause can contain complex conditions while event stream filters are more restrictive in the type of filters that can be specified. The next statement's where-clause applies the `ceil` function of the `java.lang.Math` Java library class in the where clause. The insert-into clause makes the results of the first statement available to the second statement:

```
insert into WithdrawalFiltered select * from Withdrawal where Math.ceil(amount) >= 200

select * from WithdrawalFiltered
```

4.5. Time Windows

In this section we explain the output model of statements employing a time window view and a time batch view.

4.5.1. Time Window

A time window is a moving window extending to the specified time interval into the past based on the system time. Time windows enable us to limit the number of events considered by a query, as do length windows.

As a practical example, consider the need to determine all accounts where the average withdrawal amount per account for the last 4 seconds of withdrawals is greater then 1000. The statement to solve this problem is shown below.

```
select account, avg(amount)
from Withdrawal.win:time(4 sec)
group by account
having amount > 1000
```

The next diagram serves to illustrate the functioning of a time window. For the diagram, we assume a query that simply selects the event itself and does not group or filter events.

```
select * from Withdrawal.win:time(4 sec)
```

The diagram starts at a given time `t` and displays the contents of the time window at `t + 4` and `t + 5 seconds` and so on.

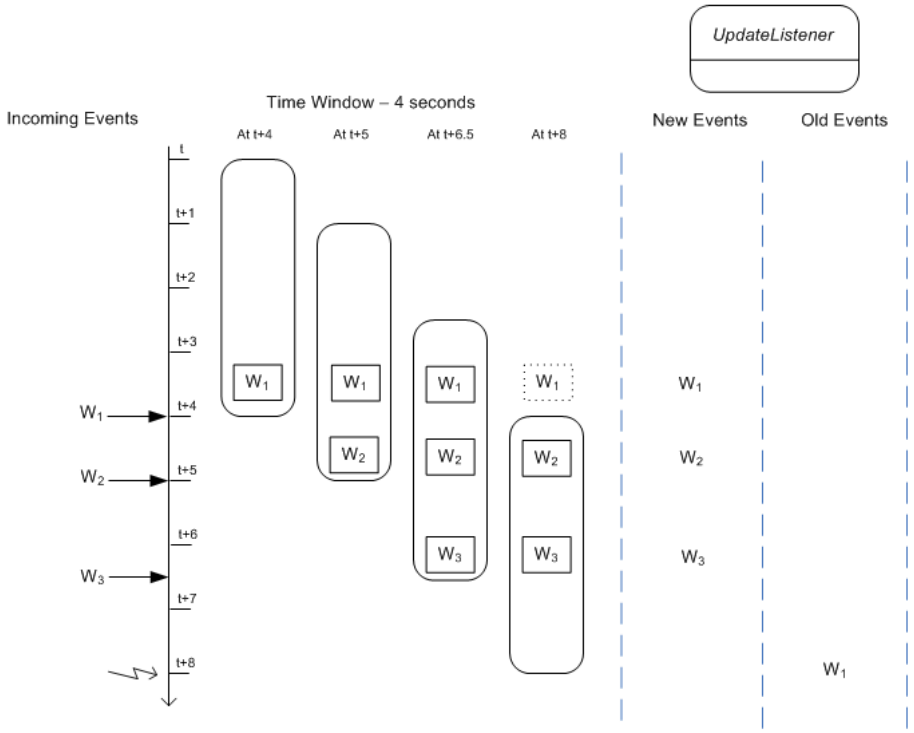


Figure 4.5. Output example for a statement with a time window

The activity as illustrated by the diagram:

1. At time `t + 4 seconds` an event W_1 arrives and enters the time window. The engine reports the new event to update listeners.

2. At time `t + 5 seconds` an event W_2 arrives and enters the time window. The engine reports the new event to update listeners.

3. At time `t + 6.5 seconds` an event W_3 arrives and enters the time window. The engine reports the new event to update listeners.

4. At time `t + 8 seconds` event W_1 leaves the time window. The engine reports the event as an old event to update listeners.

4.5.2. Time Batch

The time batch view buffers events and releases them every specified time interval in one update. Time windows control the evaluation of events, as does the length batch window.

The next diagram serves to illustrate the functioning of a time batch view. For the diagram, we assume a simple

query as below:

```
select * from Withdrawal.win:time_batch(4 sec)
```

The diagram starts at a given time `t` and displays the contents of the time window at `t + 4` and `t + 5 seconds` and so on.

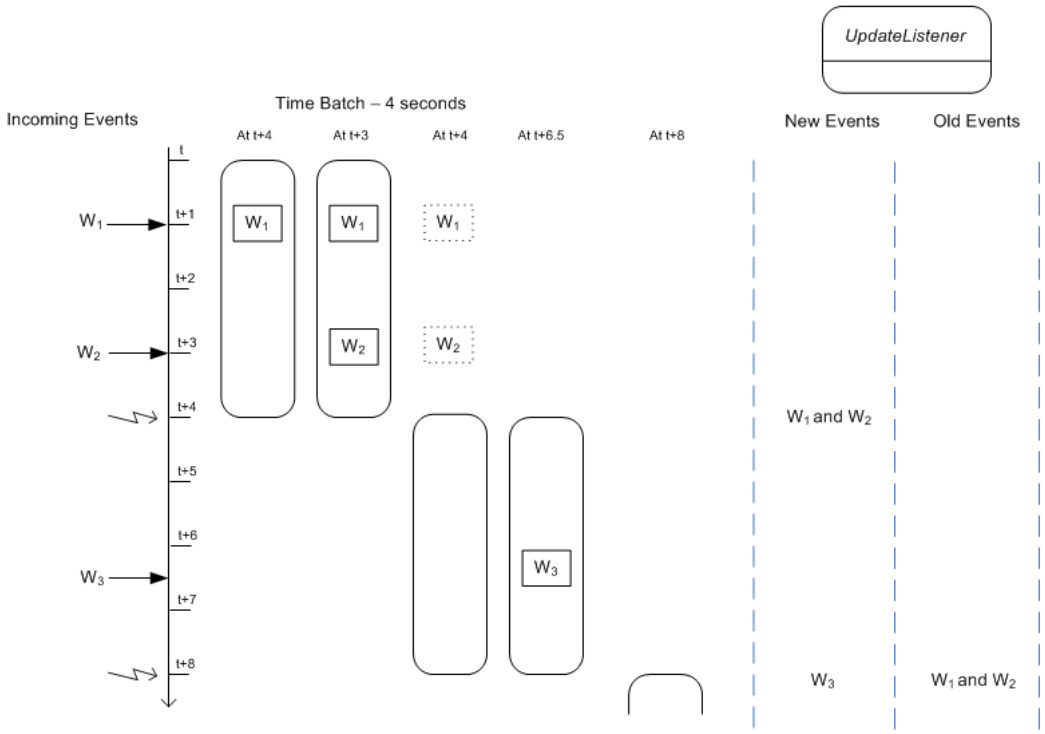


Figure 4.6. Output example for a statement with a time batch view

The activity as illustrated by the diagram:

1. At time `t + 1 seconds` an event W_1 arrives and enters the batch. No call to inform update listeners occurs.
2. At time `t + 3 seconds` an event W_2 arrives and enters the batch. No call to inform update listeners occurs.
3. At time `t + 4 seconds` the engine processes the batched events and a starts a new batch. The engine reports events W_1 and W_2 to update listeners.
4. At time `t + 6.5 seconds` an event W_3 arrives and enters the batch. No call to inform update listeners occurs.
5. At time `t + 8 seconds` the engine processes the batched events and a starts a new batch. The engine reports the event W_3 as new data to update listeners. The engine reports the events W_1 and W_2 as old data (prior batch) to update listeners.

4.6. Aggregation and Grouping

4.6.1. Insert and Remove Stream

Statements that aggregate events via aggregations functions also post remove stream events as aggregated values change.

Consider the following statement that alerts when 2 Withdrawal events have been received:

```
select count(*) as mycount from Withdrawal having count(*) = 2
```

When the engine encounters the second withdrawal event, the engine posts a new event to update listeners. The value of the "mycount" property on that new event is 2. Additionally, when the engine encounters the third Withdrawal event, it posts an old event to update listeners containing the prior value of the count. The value of the "mycount" property on that old event is also 2.

The `istream` or `rstream` keyword can be used to eliminate either new events or old events posted to listeners. The next statement uses the `istream` keyword causing the engine to call the listener only once when the second Withdrawal event is received:

```
select istream count(*) as mycount from Withdrawal having count(*) = 2
```

4.6.2. Output for Event Batches

The built-in data windows that act on batches of events are the `win:time_batch` and the `win:length_batch` views. The `win:time_batch` data window collects events arriving during a given time interval and posts collected events as a batch to listeners at the end of the time interval. The `win:length_batch` data window collects a given number of events and posts collected events as a batch to listeners when the given number of events has collected.

Let's look at how a time batch window may be used:

```
select account, amount from Withdrawal.win:time_batch(1 sec)
```

The above statement collects events arriving during a one-second interval, at the end of which the engine posts the collected events as new events (insert stream) to each listener. The engine posts the events collected during the prior batch as old events (remove stream). The engine starts posting events to listeners one second after it receives the first event and thereon.

For statements containing aggregation functions and/or a `group by` clause, the engine posts consolidated aggregation results for an event batch. For example, consider the following statement:

```
select sum(amount) as mysum from Withdrawal.win:time_batch(1 sec)
```

Following SQL (Standard Query Language) standards for queries against relational databases, the presence or absence of aggregation functions and the presence or absence of the `group by` clause dictates the number of rows posted by the engine to listeners at the end of a batch. The next sections outline the output model for batched events under aggregation and grouping.

Note that output rate limiting also generates batches of events following the output model as discussed here.

Un-aggregated and Un-grouped

An example statement for the un-aggregated and un-grouped case is as follows:

```
select * from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners one row for each event arriving during the time interval.

Fully Aggregated and Un-grouped

If your statement only selects aggregation values and does not group, your statement may look as the example below:

```
select sum(amount)
from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners a single row indicating the aggregation result. The aggregation result aggregates all events collected during the time interval.

Aggregated and Un-Grouped

If your statement selects non-aggregated properties and aggregation values, and does not group, your statement may be similar to this statement:

```
select account, sum(amount)
from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners one row per event. The aggregation result aggregates all events collected during the time interval.

Fully Aggregated and Grouped

If your statement selects aggregation values and all non-aggregated properties in the `select` clause are listed in the `group by` clause, then your statement may look similar to this example:

```
select account, sum(amount)
from Withdrawal.win:time_batch(1 sec)
group by account
```

At the end of a time interval, the engine posts to listeners one row per unique account number. The aggregation result aggregates per unique account.

Aggregated and Grouped

If your statement selects non-aggregated properties and aggregation values, and groups only some properties using the `group by` clause, your statement may look as below:

```
select account, accountName, sum(amount)
from Withdrawal.win:time_batch(1 sec)
group by account
```

At the end of a time interval, the engine posts to listeners one row per event. The aggregation result aggregates per unique account.

4.7. `EventBean` Query Results

The engine posts events to `UpdateListener` implementations as `net.esper.event.EventBean` instances. The `EventBean` represents a row (event) in your continuous query's result set.

Use the `iterator` method on `EPStatement` statements to poll or read data out of statements, if you require read-based access to statement result sets. Statement iterators also return `EventBean` instances.

The `EventBean` interface offers property type metadata via the `getEventType` method returning an `EventType`. The `EventType` provides property name, property type and underlying type information. This information can be useful to dynamically interrogate query results. The underlying event that an `EventBean` represents can be obtained via the `getUnderlying` method. Please see Chapter 5, *Event Representations* for more information on different event underlying objects.

Consider a statement that returns the symbol, count of events per symbol and average price per symbol for tick events. Our sample statement may declare a fully-qualified Java class name as the event type: `org.sample.StockTickEvent`. Assume that this class exists and exposes a `symbol` property of type String, and a `price` property of type (Java primitive) double.

```
select symbol, avg(price) as avgprice, count(*) as mycount
from org.sample.StockTickEvent
group by symbol
```

The next table summarizes the property names and types as posted by the statement above:

Table 4.1. Properties offered by sample statement aggregating price

Name	Type	Description	Java code snippet
`symbol`	java.lang.String	Value of symbol event property	`eventBean.get("symbol")`
`avgprice`	java.lang.Double	Average price per symbol	`eventBean.get("avgprice")`
`mycount`	java.lang.Long	Number of events per symbol	`eventBean.get("mycount")`

A code snippet out of a possible `UpdateListener` implementation to this statement may look as below:

```
String symbol = (String) newEvents[0].get("symbol");
Double price= (Double) newEvents[0].get("avgprice");
Long count= (Long) newEvents[0].get("mycount");
```

The engine supplies the boxed `java.lang.Double` and `java.lang.Long` types as property values rather then primitive Java types. This is because aggregated values can return a `null` value to indicate that no data is available for aggregation. Also, in a select statement that computes expressions, the underlying event objects to `EventBean` instances are of type `java.util.Map`.

Consider the next statement that specifies a wildcard selecting the same type of event:

```
select * from org.sample.StockTickEvent where price > 100
```

The property names and types provided by an `EventBean` query result row, as posted by the statement above are as follows:

Table 4.2. Properties offered by sample wildcard-select statement

Name	Type	Description	Java code snippet
`symbol`	java.lang.String	Value of symbol event property	`eventBean.get("symbol")`
`price`	double	Value of price event property	`eventBean.get("price")`

As an alternative to querying individual event properties via the `get` methods, the `getUnderlying` method on `EventBean` returns the underlying object representing the query result. In the sample statement that features a wildcard-select, the underlying event object is of type `org.sample.StockTickEvent`:

```
StockTickEvent tick = (StockTickEvent) newEvents[0].getUnderlying();
```

Chapter 5. Event Representations

5.1. Event Underlying Java Objects

An event is an immutable record of a past occurrence of an action or state change. An event can have a set of event properties that supply information about the event. An event also has an underlying Java object type.

In Esper, an event can be represented by any of the following underlying Java objects:

Table 5.1. Event Underlying Java Objects

Java Class	Description
`java.lang.Object`	Any Java POJO (plain-old java object) with getter methods following JavaBean conventions; Also legacy Java classes not following JavaBean conventions
`java.util.Map`	Map events are key-values pairs
`org.w3c.dom.Node`	XML document object model (DOM)

5.2. Event Properties

Esper expressions can include simple as well as indexed, mapped and nested event properties. The table below outlines the different types of properties and their syntax in an event expression. This syntax allows statements to query deep JavaBean objects graphs, XML structures and Map events.

Table 5.2. Types of Event Properties

Type	Description	Syntax	Example
Simple	A property that has a single value that may be retrieved.	`name`	`sensorId`
Indexed	An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript).	`name[index]`	`sensor[0]`
Mapped	A mapped property stores a keyed collection of objects (all of the same type).	`name('key')`	`sensor('light')`
Nested	A nested property is a property that lives within another property of an event.	`name.nestedname`	`sensor.value`

Combinations are also possible. For example, a valid combination could be `person.address('home').street[0]`.

5.3. Plain Java Object Events

Plain Java object events are object instances that expose event properties through JavaBean-style getter methods. Events classes or interfaces 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.

Esper supports JavaBean-style event classes that extend a superclass or implement one or more interfaces. Also, Esper event pattern and EQL statements can refer to Java interface classes and abstract classes.

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

The `hashCode` and `equals` methods do not need to be implemented. The implementation of these methods by a Java event class does not affect the behavior of the engine in any way.

Please see Chapter 2, *Configuration* on options for naming event types represented by Java object event classes.

5.3.1. Java Object Event Properties

As outlined earlier, the different property types are supported by the standard JavaBeans specification, and some of which are uniquely supported by Esper:

- *Simple* properties have a single value that may be retrieved. The underlying property type might be a Java language primitive (such as int, a simple object (such as a java.lang.String), or a more complex object whose class is defined either by the Java language, by the application, or by a class library included with the application.
- *Indexed* - An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript). Alternatively, the entire set of values may be retrieved using an array.
- *Mapped* - As an extension to standard JavaBeans APIs, Esper considers any property that accepts a String-valued key a mapped property.
- *Nested* - A nested property is a property that lives within another Java object which itself is a property of an event.

Assume there is an EmployeeEvent event class as shown below. The mapped and indexed properties in this example return Java objects but could also return Java language primitive types (such as int or String). The Address object and Employee objects can themselves have properties that are nested within them, such as a streetName in the Address object or a name of the employee in the Employee object.

```
public class EmployeeEvent {
        public String getFirstName();
        public Address getAddress(String type);
        public Employee getSubordinate(int index);
        public Employee[] getAllSubordinates();
}
```

Simple event properties require a getter-method that returns the property value. In this example, the `getFirstName` getter method returns the `firstName` event property of type String.

Indexed event properties require either one of the following getter-methods. A method that takes an integer-type key value and returns the property value, such as the `getSubordinate` method. Or a method that returns an array-type such as the `getSubordinates` getter method, which returns an array of Employee. In an EQL or event pattern statement, indexed properties are accessed via the `property[index]` syntax.

Mapped event properties require a getter-method that takes a String-typed key value and returns the property value, such as the `getAddress` method. In an EQL or event pattern statement, mapped properties are accessed via the `property('key')` syntax.

Nested event properties require a getter-method that returns the nesting object. The `getAddress` and `getSubordinate` methods are mapped and indexed properties that return a nesting object. In an EQL or event pattern statement, nested properties are accessed via the `property.nestedProperty` syntax.

All event pattern and EQL statements allow the use of indexed, mapped and nested properties (or a combination of these) anywhere where one or more event property names are expected. The below example shows different combinations of indexed, mapped and nested properties in filters of event pattern expressions.

```
every EmployeeEvent(firstName='myName')
every EmployeeEvent(address('home').streetName='Park Avenue')
every EmployeeEvent(subordinate[0].name='anotherName')
every EmployeeEvent(allSubordinates[1].name='thatName')
every EmployeeEvent(subordinate[0].address('home').streetName='Water Street')
```

Similarly, the syntax can be used in EQL statements in all places where an event property name is expected, such as in select lists, where-clauses or join criteria.

```
select firstName, address('work'), subordinate[0].name, subordinate[1].name
from EmployeeEvent
where address('work').streetName = 'Park Ave'
```

Event properties that are enumeration values can be compared by means of the Enumeration `valueOf` method on the `java.lang.Enum` class. An example could look as follows:

```
every MyEvent(enumProp=EnumClass.valueOf('ENUM_VALUE_1'))
```

Java classes that do not follow JavaBean conventions, such as legacy Java classes that expose public fields, or methods not following naming conventions, require additional configuration. Via configuration it is also possible to control case sensitivity in property name resolution. The relevant section in the chapter on configuration is Section 2.4.1.2, “Non-JavaBean and Legacy Java Event Classes”.

5.4. `java.util.Map` Events

Events can also be represented by objects that implement the `java.util.Map` interface. Event properties of `Map` events are the values in the map accessible through the `get` method exposed by the `java.util.Map` interface.

The engine can process `java.util.Map` events via the `sendEvent(Map map, String eventTypeAlias)` method on the `EPRuntime` interface. Entries in the Map represent event properties. Keys must be of type `java.util.String` for the engine to be able to look up event property names specified by pattern or EQL statements. Values can be of any type. JavaBean-style objects as values in a `Map` can also be processed by the engine.

In order to use `Map` events, the event type name and property names and types must be made known to the engine via Configuration. Please see the examples in Section 2.4.2, “Events represented by java.util.Map”.

The code snippet below creates and processes a `Map` event. The example assumes the `CarLocationUpdateEvent` event type alias has been configured.

```
Map event = new HashMap();
event.put("carId", carId);
event.put("direction", direction);
```

```
epRuntime.sendEvent(event, "CarLocUpdateEvent");
```

The `CarLocUpdateEvent` can now be used in a statement:

```
select carId from CarLocUpdateEvent.win:time(1 min) where direction = 1
```

The engine can also query Java objects as values in a `Map` event via the nested property syntax. Thus `Map` events can be used to aggregate multiple datastructures into a single event and query the composite information in a convenient way. The example below demonstrates a `Map` event with a transaction and an account object.

```
Map event = new HashMap();
event.put("txn", txn);
event.put("account", account);
epRuntime.sendEvent(event, "TxnEvent");
```

An example statement could look as follows.

```
select account.id, account.rate * txn.amount from TxnEvent.win:time(60 sec) group by account.id
```

5.5. `org.w3c.dom.Node` XML Events

Events can also be represented as `org.w3c.dom.Node` instances and send into the engine via the `sendEvent` method on `EPRuntime`. Please note that configuration is required for allowing the engine to map the event type alias to `Node` element names. See Chapter 2, *Configuration*.

Esper allows configuring XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name by which their result values will be available for use in expressions. For XML documents that follow an XML schema, Esper can load and interrogate your schema and validate event property names and types against the schema information.

Nested, mapped and indexed event properties are also supported in expressions against `org.w3c.dom.Node` events. Thus XML trees can conveniently be interrogated using the existing event property syntax for querying JavaBean objects, JavaBean object graphs or `java.util.Map` events.

Let's look at how a sample XML document could be queried, given the sample XML below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Sensor>
        <ID>urn:epc:1:4.16.36<ID>
        <Observation Command="READ_PALLET_TAGS_ONLY">
                <ID>00000001<ID>
                <Tag>
                        <ID>urn:epc:1:2.24.400<ID>
                </Tag>
                <Tag>
                        <ID>urn:epc:1:2.24.401<ID>
                </Tag>
        </Observation>
</Sensor>
```

To configure the engine for processing Sensor documents, simply configure a `SensorEvent` event type alias for the `Sensor` element name via Configuration. Now the document can be queried as below.

```
select ID, Observation.ID, Observation.Command, Observation.Tag[0], countTags
from SensorEvent.win:time(30 sec)
```

The equivalent XPath expressions to each of the properties are listed below.

- The equivalent XPath expression to `Observeration.ID` is `/Sensor/Observation/ID`
- The equivalent XPath expression to `Observeration.Command` is `/Sensor/Observation/@Command`
- The equivalent XPath expression to `Observeration.Tag[0]` is `/Sensor/Observation/Tag[position() = 1]`
- The equivalent XPath expression to `countTags` is `count(/Sensor/Observation/Tag)` for returning a count of tag elements. This assumes the `countTags` property has been configured as an XPath property.

By specifying an event property such below:

```
nestedElement.mappedElement('key').indexedElement[1]
```

The equivalent XPath expression is as follows:

```
/simpleEvent/nestedElement/mappedElement[@id='key']/indexedElement[position() = 2]
```

Chapter 6. EQL Reference

6.1. EQL Introduction

EQL statements are used to derive and aggregate information from one or more streams of events, and to join or merge 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 its 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:length`, `win:length_batch`, `win:time`, `win:time_batch`, `win:ext_time`, `ext:sort_window`
2. Views for aggregation: `std:unique`, `std:groupby`, `std:lastevent` (note: the `group-by` clause and the `std:groupby` view are very similar in function, see view description for differences)
3. Views that derive statistics: `std:size`, `stat:uni`, `stat:linest`, `stat:correl`, `stat:weighted_avg`, `stat:cube`

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.

```
select avg(price) from StockTick.win:time(30 sec) where symbol='IBM'
```

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, may contain a where clause with search conditions and so on.

```
[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]
```

6.2.1. Specifying Time Periods

Time-based windows as well as pattern observers and guards take a time period as a parameter. Time periods follow the syntax below.

```
time-period : [day-part] [hour-part] [minute-part] [seconds-part] [milliseconds-part]

day-part : number ("days" | "day")
hour-part : number ("hours" | "hour")
minute-part : number ("minutes" | "minute" | "min")
seconds-part : number ("seconds" | "second" | "sec")
milliseconds-part : number ("milliseconds" | "millisecond" | "msec")
```

Some examples of time periods are:

```
10 seconds
10 minutes 30 seconds
20 sec 100 msec
1 day 2 hours 20 minutes 15 seconds 110 milliseconds
0.5 minutes
```

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.

The select clause also offers optional `istream` and `rstream` keywords to control how events are posted to `UpdateListener` instances listening to the statement.

The syntax for the `select` clause is summarized below.

```
select [rstream | istream] * | expression_list ...
```

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 univariate statistics for the last 30 seconds of IBM stock ticks for price.

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

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

The `*` wildcard and expressions can also be combined in a `select` clause. The combination selects all event properties and in addition the computed values as specified by any additional expressions that are part of the `select` clause. Here is an example that selects all properties of stock tick events plus a computed product of

price and volume that the statement names 'pricevolume':

```
select *, price * volume as pricevolume from StockTick(symbol='IBM')
```

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 sec)
```

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 property.

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

6.3.5. Selecting `istream` and `rstream` events

The optional `istream` and `rstream` keywords in the select clause define the event stream posted to listeners to the statement.

If neither keyword is specified, the engine posts insert stream events via the `newEvents` parameter to the `update` method of `UpdateListener` instances listening to the statement. The engine posts remove stream events to the `oldEvents` parameter of the `update` method. The insert stream consists of the events entering the respective window(s) or stream(s) or aggregations, while the remove stream consists of the events leaving the respective window(s) or the changed aggregation result. See Chapter 4, *Understanding the Output Model* for more information on insert and remove streams.

By specifying the `istream` keyword you can instruct the engine to only post insert stream events via the `newEvents` parameter to the `update` method on listeners. The engine will then not post any remove stream

events, and the `oldEvents` parameter is always a null value.

By specifying the `rstream` keyword you can instruct the engine to only post remove stream events via the `newEvents` parameter to the `update` method on listeners. The engine will then not post any insert stream events, and the `oldEvents` parameter is also always a null value.

The following statement selects only the events that are leaving the 30 second time window.

```
select rstream * from StockTick.win:time(30 sec)
```

The `istream` and `rstream` keywords in the select clause are matched by same-name keywords available in the insert into clause. While the keywords in the select clause control the event stream posted to listeners to the statement, the same keywords in the insert into clause specify the event stream that the engine makes available to other statements.

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 can consists of either a filter-based event stream definition or a pattern-based event stream definition.

For joins and outer joins, specify two or more event streams. Joins between pattern-based and filter-based event streams are also supported.

Esper supports joins against relational databases for access to historical or reference data as explained in Section 6.13, “Joining Relational Data via SQL”.

6.4.1. Filter-based event streams

For filter-based event streams, the event stream definition *stream_def* as shown in the from-clause syntax consists of an event type, optional filter expressions and an optional list of views that derive data from a stream. The syntax for a filter-based event stream is as below:

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

The following EQL statement shows event type, filter criteria and views combined in one statement. It 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 Java class name `org.esper.example.StockTick`. The expression filters for events where the property `symbol` has a 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 volumeStats
```

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. In the above example, events with a symbol other then IBM do not enter the time window.

Specifying an event type

The simplest form of filter is a filter for events of a given type without any conditions on the event property values. This filter matches any event of that type regardless of the event's properties. The example below is such a filter.

```
select * from com.mypackage.myevents.RfidEvent
```

Instead of the fully-qualified Java class name any other event name can be mapped via Configuration to a Java class, making the resulting statement more readable:

```
select * from RfidEvent
```

Interfaces and superclasses are also supported as event types. In the below example `IRfidReadable` is an interface class.

```
select * from org.myorg.rfid.IRfidReadable
```

Specifying filter criteria

The filtering criteria to filter for events with certain event property values are placed within parenthesis after the event type name:

```
select * from RfidEvent(category="Perishable")
```

All expressions can be used in filters, including static methods that return a boolean value:

```
select * from RfidEvent(MyRFIDLib.isInRange(x, y) or (x < 0 and y < 0))
```

Filter expressions can be separated via a single comma ',' . The comma represents a logical AND between filter expressions:

```
select * from RfidEvent(zone=1, category=10)
...is equivalent to...
select * from RfidEvent(zone=1 and category=10)
```

The following set of operators are highly optimized through indexing and are the preferred means of filtering in high-volumne event streams:

- equals `=`
- not equals `!=`
- comparison operators `< , > , >=, <=`
- ranges
  - use the `between` keyword for a closed range where both endpoints are included
  - use the `in` keyword and round `()` or square brackets `[]` to control how endpoints are included
  - for inverted ranges use the `not` keyword and the `between` or `in` keywords
- list-of-values checks using the `in` keyword or the `not in` keywords followed by a comma-separated list of values

At compile time as well as at run time, the engine scans new filter expressions for sub-expressions that can be indexed. Indexing filter values to match event properties of incoming events enables the engine to match incoming events faster. The above list of operators represents the set of operators that the engine can best convert into indexes. The use of comma or logical `and` in filter expressions does not impact optimizations by the engine.

Filtering Ranges

Ranges come in the following 4 varieties. The use of round `()` or square `[]` bracket dictates whether an endpoint is included or excluded. The low point and the high-point of the range are separated by the colon `:` character.

- Open ranges that contain neither endpoint `(low:high)`
- Closed ranges that contain both endpoints `[low:high]`. The equivalent 'between' keyword also defines a closed range.
- 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]`

The next statement shows a filter specifying a range for `x` and `y` values of RFID events. The range includes both endpoints therefore uses `[]` hard brackets.

```
mypackage.RfidEvent(x in [100:200], y in [0:100])
```

The `between` keyword is equivalent for closed ranges. The same filter using the `between` keyword is:

```
mypackage.RfidEvent(x between 100 and 200, y between 0 and 50)
```

The `not` keyword can be used to determine if a value falls outside a given range:

```
mypackage.RfidEvent(x not in [0:100])
```

The equivalent statement using the `between` keyword is:

```
mypackage.RfidEvent(x not between 0 and 100)
```

Filtering Sets of Values

The `in` keyword for filter criteria determines if a given value matches any value in a list of values.

In this example we are interested in RFID events where the category matches any of the given values:

```
mypackage.RfidEvent(category in ('Perishable', 'Container'))
```

By using the `not in` keywords we can filter events with a property value that does not match any of the values in a list of values:

```
mypackage.RfidEvent(category not in ('Household', 'Electrical'))
```

Filter Limitations

The following restrictions apply to filter criteria:

- Range and comparison operators require the event property to be of a numeric type.
- Aggregation functions are not allowed within filter expressions.
- The `prev` previous event function and the `prior` prior event function cannot be used in filter expressions.

6.4.2. Pattern-based event streams

Event pattern expressions can also be used to specify one or more event streams in an EQL statement. For pat-

tern-based event streams, the event stream definition *stream_def* consists of the keyword `pattern` and a pattern expression in brackets `[]`. The syntax for an event stream definition using a pattern expression is below. As in filter-based event streams, an optional list of views that derive data from the stream can be supplied.

```
pattern [pattern_expression] [.view_spec] [.view_spec] [...]
```

The next statement specifies an event stream that consists of both stock tick events and trade events. The example tags stock tick events with the name "tick" and trade events with the name "trade".

```
select * from pattern [every tick=StockTickEvent or every trade=TradeEvent]
```

This statement generates an event every time the engine receives either one of the event types. The generated events resemble a map with "tick" and "trade" keys. For stock tick events, the "tick" key value is the underlying stock tick event, and the "trade" key value is a null value. For trade events, the "trade" key value is the underlying trade event, and the "tick" key value is a null value.

Lets further refine this statement adding a view the gives us the last 30 seconds of either stock tick or trade events. Lets also select prices and a price total.

```
select tick.price as tickPrice, trade.price as tradePrice,
       sum(tick.price) + sum(trade.price) as total
  from pattern [every tick=StockTickEvent or every trade=TradeEvent].win:time(30 sec)
```

Note that in the statement above `tickPrice` and `tradePrice` can each be null values depending on the event processed. Therefore, an aggregation function such as `sum(tick.price + trade.price))` would always return null values as either of the two price properties are always a null value for any event matching the pattern. Use the `coalesce` function to handle null values, for example: `sum(coalesce(tick.price, 0) + coalesce(trade.price, 0))`.

6.4.3. Specifying views

Views are used to derive or aggregate data. Views can be staggered onto each other. See the section Section 6.16, “Built-in views” on the views available.

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

The below example serves to show views and staggering of views. It uses a car location event that contains information about the location of a car on a highway.

The first view `std:groupby('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:groupby({'expressway', 'direction', 'segment'})` groups each event by its 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.

```
select * from CarLocEvent.std:groupby('carId').win:length(4).
  std:groupby({'expressway', 'direction', 'segment'}).std:size()
```

6.5. Specifying Search Conditions: the *Where* Clause

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

Comparison operators `=, < , > , >=, <=, !=, <>, is null, is not null` and logical combinations via `and` and `or` are supported in the where clause. The where clause can also introduce join conditions as outlined in Section 6.10, “Joining Event Streams”. Where-clauses can also contain expressions. Some examples are listed below.

```
...where fraud.severity = 5 and amount > 500
...where (orderItem.orderId is null) or (orderItem.class != 10)
...where (orderItem.orderId = null) or (orderItem.class <> 10)
...where itemCount / packageCount > 10
```

6.6. Aggregates and grouping: the *Group-by* Clause and the *Having* Clause

6.6.1. Using aggregate functions

The aggregate functions are `sum, avg, count, max, min, median, stddev, avedev`. You can use aggregate functions to calculate and summarize data from event properties. For example, to find out the total price for all stock tick events in the last 30 seconds, type:

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

Here is the syntax for aggregate functions:

```
aggregate_function( [all | distinct] expression)
```

You can apply aggregate functions to all events in an event stream window or other view, or to one or more groups of events. From each set of events to which an aggregate function is applied, Esper generates a single value.

`Expression` is usually an event property name. However it can also be a constant, function, or any combination of event property names, constants, and functions connected by arithmetic operators.

For example, to find out the average price for all stock tick events in the last 30 seconds if the price was doubled:

```
select avg(price * 2) from StockTickEvent.win:time(30 seconds)
```

You can use the optional keyword `distinct` with all aggregate functions to eliminate duplicate values before the aggregate function is applied. The optional keyword `all` which performs the operation on all events is the default.

The syntax of the aggregation functions and the results they produce are shown in below table.

Table 6.1. Syntax and results of aggregate functions

Aggregate Function	Result
sum([all|distinct] *expression*)	Totals the (distinct) values in the expression, returning a value of `long`,

Aggregate Function	Result
	`double, float or integer` type depending on the expression
avg([all|distinct] *expression*)	Average of the (distinct) values in the expression, returning a value of `double` type
count([all|distinct] *expression*)	Number of the (distinct) non-null values in the expression, returning a value of `long` type
count(*)	Number of events, returning a value of `long` type
max([all|distinct] *expression*)	Highest (distinct) value in the expression, returning a value of the same type as the expression itself returns
min([all|distinct] *expression*)	Lowest (distinct) value in the expression, returning a value of the same type as the expression itself returns
median([all|distinct] *expression*)	Median (distinct) value in the expression, returning a value of `double` type
stddev([all|distinct] *expression*)	Standard deviation of the (distinct) values in the expression, returning a value of `double` type
avedev([all|distinct] *expression*)	Mean deviation of the (distinct) values in the expression, returning a value of `double` type

You can use aggregation functions in a `select` clause and in a `having` clause. You cannot use aggregate functions in a `where` clause, but you can use the where clause to restrict the events to which the aggregate is applied. The next query computes the average and sum of the price of stock tick events for the symbol IBM only, for the last 10 stock tick events regardless of their symbol.

```
select 'IBM stats' as title, avg(price) as avgPrice, sum(price) as sumPrice
from StockTickEvent.win:length(10)
where symbol='IBM'
```

In the above example the length window of 10 elements is not affected by the where-clause, i.e. all events enter and leave the length window regardless of their symbol. If we only care about the last 10 IBM events, we need to add filter criteria as below.

```
select 'IBM stats' as title, avg(price) as avgPrice, sum(price) as sumPrice
from StockTickEvent(symbol='IBM').win:length(10)
where symbol='IBM'
```

You can use aggregate functions with any type of event property or expression, with the following exceptions:

1. You can use `sum, avg, median, stddev, avedev` with numeric event properties only

Esper ignores any null values returned by the event property or expression on which the aggregate function is operating, except for the `count(*)` function, which counts null values as well. All aggregate functions return null if the data set contains no events, or if all events in the data set contain only null values for the aggregated

expression.

6.6.2. Organizing statement results into groups: the *Group-by* clause

The `group by` clause is optional in all EQL statements. The `group by` clause divides the output of an EQL statement into groups. You can group by one or more event property names, or by the result of computed expressions. When used with aggregate functions, `group by` retrieves the calculations in each subgroup. You can use `group by` without aggregate functions, but generally that can produce confusing results.

For example, the below statement returns the total price per symbol for all stock tick events in the last 30 seconds:

```
select symbol, sum(price) from StockTickEvent.win:time(30 sec) group by symbol
```

The syntax of the group by clause is:

```
group by arregate_free_expression [, arregate_free_expression] [, ...]
```

Esper places the following restrictions on expressions in the `group by` clause:

1. Expressions in the `group by` cannot contain aggregate functions
2. Event properties that are used within aggregate functions in the `select` clause cannot also be used in a `group by` expression

You can list more then one expression in the `group by` clause to nest groups. Once the sets are established with `group by` the aggregation functions are applied. This statement posts the median volume for all stock tick events in the last 30 seconds per symbol and tick data feed. Esper posts one event for each group to statement listeners:

```
select symbol, tickDataFeed, median(volume)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

In the statement above the event properties in the `select` list (symbol, tickDataFeed) are also listed in the `group by` clause. The statement thus follows the SQL standard which prescribes that non-aggregated event properties in the `select` list must match the `group by` columns.

Esper also supports statements in which one or more event properties in the `select` list are not listed in the `group by` clause. The statement below demonstrates this case. It calculates the standard deviation for the last 30 seconds of stock ticks aggregating by symbol and posting for each event the symbol, tickDataFeed and the standard deviation on price.

```
select symbol, tickDataFeed, stddev(price) from StockTickEvent.win:time(30 sec) group by symbol
```

The above example still aggregates the `price` event property based on the `symbol`, but produces one event per incoming event, not one event per group.

Additionally, Esper supports statements in which one or more event properties in the `group by` clause are not listed in the `select` list. This is an example that calculates the mean deviation per `symbol` and `tickDataFeed` and posts one event per group with `symbol` and mean deviation of price in the generated events. Since tickDataFeed is not in the posted results, this can potentially be confusing.

```
select symbol, avedev(price)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

Expressions are also allowed in the `group by` list:

```
select symbol * price, count(*) from StockTickEvent.win:time(30 sec) group by symbol * price
```

If the `group by` expression resulted in a null value, the null value becomes its own group. All null values are aggregated into the same group. If you are using the `count(expression)` aggregate function which does not count null values, the count returns zero if only null values are encountered.

You can use a `where` clause in a statement with `group by`. Events that do not satisfy the conditions in the `where` clause are eliminated before any grouping is done. For example, the statement below posts the number of stock ticks in the last 30 seconds with a volume larger then 100, posting one event per group (symbol).

```
select symbol, count(*) from StockTickEvent.win:time(30 sec) where volume > 100 group by symbol
```

6.6.3. Selecting groups of events: the *Having* clause

Use the `having` clause to pass or reject events defined by the `group-by` clause. The `having` clause sets conditions for the `group by` clause in the same way `where` sets conditions for the `select` clause, except `where` cannot include aggregate functions, while `having` often does.

This statement is an example of a `having` clause with an aggregate function. It posts the total price per symbol for the last 30 seconds of stock tick events for only those symbols in which the total price exceeds 1000. The `having` clause eliminates all symbols where the total price is equal or less then 1000.

```
select symbol, sum(price)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000
```

To include more then one condition in the `having` clause combine the conditions with `and`, `or` or `not`. This is shown in the statement below which selects only groups with a total price greater then 1000 and an average volume less then 500.

```
select symbol, sum(price), avg(volume)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000 and avg(volume) < 500
```

Esper places the following restrictions on expressions in the `having` clause:

1. Any expressions that contain aggregate functions must also occur in the `select` clause

A statement with the `having` clause should also have a `group by` clause. If you omit `group-by`, all the events not excluded by the `where` clause return as a single group. In that case `having` acts like a `where` except that `having` can have aggregate functions.

The `having` clause can also be used without `group by` clause as the below example shows. The example below posts events where the price is less then the current running average price of all stock tick events in the last 30 seconds.

```
select symbol, price, avg(price)
from StockTickEvent.win:time(30 sec)
having price < avg(price)
```

6.6.4. How the stream filter, *Where*, *Group By* and *Having* clauses interact

When you include filters, the `where` condition, the `group by` clause and the `having` condition in an EQL statement the sequence in which each clause affects events determines the final result:

1. The event stream's filter condition, if present, dictates which events enter a window (if one is used). The filter discards any events not meeting filter criteria.
2. The `where` clause excludes events that do not meet its search condition.
3. Aggregate functions in the select list calculate summary values for each group.
4. The `having` clause excludes events from the final results that do not meet its search condition.

The following query illustrates the use of filter, `where`, `group by` and `having` clauses in one statement with a `select` clause containing an aggregate function.

```
select tickDataFeed, stddev(price)
from StockTickEvent(symbol='IBM').win:length(10)
where volume > 1000
group by tickDataFeed
having stddev(price) > 0.8
```

Esper filters events using the filter criteria for the event stream `StockTickEvent`. In the example above only events with symbol IBM enter the length window over the last 10 events, all other events are simply discarded. The `where` clause removes any events posted by the length window (events entering the window and event leaving the window) that do not match the condition of volume greater then 1000. Remaining events are applied to the `stddev` standard deviation aggregate function for each tick data feed as specified in the `group by` clause. Each `tickDataFeed` value generates one event. Esper applies the `having` clause and only lets events pass for `tickDataFeed` groups with a standard deviation of price greater then 0.8.

6.6.5. Comparing the *Group By* clause and the *std:groupby* view

The *group by* clause as well as the built-in *std:groupby* view are similar in their ability to group events. This section explains the key differences in their behavior and use.

The *group by* clause works together with aggregation functions in your statement to produce an aggregation result per group. In greater detail, this means that when a new event arrives, the engine applies the expressions in the *group by* clause to determine a grouping key. If the engine has not encountered that grouping key before (a new group), the engine creates a set of new aggregation results for that grouping key and performs the aggregation changing that new set of aggregation results. If the grouping key points to an existing set of prior aggregation results (an existing group), the engine performs the aggregation changing the prior set of aggregation results for that group.

The *std:groupby* view is a built-in view that also groups events. The view is described in greater detail in Section 6.16.2.2, “Group By (std:groupby)”. Its primary use is to create a separate data window per group, or more generally to create separate instances of all its sub-views for each grouping key encountered.

The next example shows two queries that produce equivalent results. The query using the *group by* clause is generally preferable as it can be easier to read:

```
select symbol, sum(price) from StockTickEvent group by symbol
// ... is equivalent to ...
select symbol, sum(price) from StockTickEvent.std:groupby('symbol')
```

As a side note, by adding an output rate limiting clause to a statement that contains a *group by* clause we can control output of groups to obtain one row for each group, generating an event per group at the given output

frequency:

```
select symbol, sum(price) from StockTickEvent group by symbol output every 5 seconds
```

The next example shows two queries that are NOT equivalent as the length window is ungrouped in the first query, and grouped in the second query:

```
select symbol, sum(price) from StockTickEvent.win:length(10) group by symbol
// ... NOT equivalent to ...
select symbol, sum(price) from StockTickEvent.std:groupby('symbol').win:length(10)
```

The key difference between the two statements is that in the first statement the length window is ungrouped and applies to all events regardless of group. While in the second query each group gets its own instance of a length window. For example, in the second query events arriving for symbol "ABC" get a length window of 10 events, and events arriving for symbol "DEF" get their own length window of 10 events.

6.7. Stabilizing and Limiting Output: the *Output* Clause

6.7.1. Output Clause Options

The `output` clause is optional in Esper and is used to control or stabilize the rate at which events are output. For example, the following statement batches old and new events and outputs them at the end of every 90 second interval.

```
select * from StockTickEvent.win:length(5) output every 90 seconds
```

Here is the syntax for output rate limiting:

```
output [all | first | last] every number [minutes | seconds | events]
```

The `all` keyword is the default and specifies that all events in a batch should be output. The batch size can be specified in terms of time or number of events.

The `first` keyword specifies that only the first event in an output batch is to be output. Using the `first` keyword instructs the engine to output the first matching event as soon as it arrives, and then ignore matching events for the time interval or number of events specified. After the time interval elapsed, or the number of matching events has been reached, the next first matching event is output again and the following interval the engine again ignores matching events.

The `last` keyword specifies to only output the last event at the end of the given time interval or after the given number of matching events have been accumulated.

The time interval can also be specified in terms of minutes; the following statement is identical to the first one.

```
select * from StockTickEvent.win:length(5) output every 1.5 minutes
```

A second way that output can be stabilized is by batching events until a certain number of events have been collected. The next statement only outputs when either 5 (or more) new or 5 (or more) old events have been batched.

```
select * from StockTickEvent.win:time(30 sec) output every 5 events
```

Additionally, event output can be further modified by the optional `last` keyword, which causes output of only

the last event to arrive into an output batch.

```
select * from StockTickEvent.win:time(30 sec) output last every 5 events
```

Using the `first` keyword you can be notified at the start of the interval. The allows to watch for situations such as a rate falling below a threshold and only be informed every now and again after the specified output interval, but be informed the moment it first happens.

```
select * from TickRate.win:time(30 seconds) where rate<100 output first every 60 seconds
```

6.7.2. Group By, Having and Output clause interaction

The `output` clause interacts in two ways with the `group by` and `having` clauses. First, in the `output every n events` case, the number `n` refers to the number of events arriving into the `group by clause`. That is, if the `group by` clause outputs only 1 event per group, or if the arriving events don't satisfy the `having` clause, then the actual number of events output by the statement could be fewer than `n`.

Second, the `last` and `all` keywords have special meanings when used in a statement with aggregate functions and the `group by` clause. The `last` keyword specifies that only groups whose aggregate values have been updated with the most recent batch of events should be output. The `all` keyword (the default) specifies that the most recent data for all groups seen so far should be output, whether or not these groups' aggregate values have just been updated.

6.8. Sorting Output: the *Order By* Clause

The `order by` clause is optional in Esper. It is used for ordering output events by their properties, or by expressions involving those properties. For example, the following statement outputs batches of 5 or more stock tick events that are sorted first by price and then by volume.

```
select symbol from StockTickEvent.win:time(60 sec)
output every 5 events
order by price, volume
```

Here is the syntax for the `order by` clause:

```
order by expression [asc | desc] [, expression [asc | desc]] [, ...]
```

Esper places the following restrictions on the expressions in the `order by` clause:

1. All aggregate functions that appear in the `order by` clause must also appear in the `select` expression.

Otherwise, any kind of expression that can appear in the `select` clause, as well as any alias defined in the `select` clause, is also valid in the order by clause.

6.9. Merging Streams and Continuous Insertion: the *Insert Into* Clause

The `insert into` clause is optional in Esper. This clause can be specified to make the results of a statement available as an event stream for use in further statements. The clause can also be used to merge multiple event streams to form a single stream of events.

```
insert into CombinedEvent
```

```
select A.customerId as custId, A.timestamp - B.timestamp as latency
  from EventA.win:time(30 min) A, EventB.win:time(30 min) B
 where A.txnId = B.txnId
```

The `insert into` clause in above statement generates events of type `CombinedEvent`. Each generated `CombinedEvent` event has 2 event properties named "custId" and "latency". The events generated by above statement can be used in further statements. The below statement uses the generated events.

```
select custId, sum(latency)
  from CombinedEvent.win:time(30 min)
 group by custId
```

The `insert into` clause can consist of just an event type alias, or of an event type alias and 1 or more event property names. The syntax for the `insert into` clause is as follows:

```
insert [istream | rstream] into event_type_alias  [ (property_name [, property_name] ) ]
```

The `istream` (default) and `rstream` keywords are optional. If neither keyword or the `istream` keyword is specified, the engine supplies the insert stream events generated by the statement. The insert stream consists of the events entering the respective window(s) or stream(s). If the `rstream` keyword is specified, the engine supplies the remove stream events generated by the statement. The remove stream consists of the events leaving the respective window(s).

The `event_type_alias` is an identifier that names the events generated by the engine. The identifier can be used in statements to filter and process events of the given name.

The engine also allows listeners to be attached to a statement that contain an `insert into` clause.

To merge event streams, simply use the same `event_type_alias` identifier in all EQL statements that merge their result event streams. Make sure to use the same number and names of event properties and event property types match up.

Esper places the following restrictions on the `insert into` clause:

1. The number of elements in the `select` clause must match the number of elements in the `insert into` clause if the clause specifies a list of event property names
2. If the event type alias has already been defined by a prior statement or configuration, and the event property names and types do not match, an exception is thrown at statement creation time.

The example statement below shows the alternative form of the `insert into` clause that explicitly defines the property names to use.

```
insert into CombinedEvent (custId, latency)
select A.customerId, A.timestamp - B.timestamp
...
```

The `rstream` keyword can be useful to indicate to the engine to generate only remove stream events. This can be useful if we want to trigger actions when events leave a window rather then when events enter a window. The statement below generates `CombinedEvent` events when EventA and EventB leave the window after 30 minutes (1800 seconds).

```
insert rstream into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as latency
  from EventA.win:time(30 min) A, EventB.win:time(30 min) B
 where A.txnId = B.txnId
```

6.10. Joining Event Streams

Two or more event streams can be part of the `from` clause and thus both streams determine the resulting events. The where-clause lists the join conditions that Esper uses to relate events in the two or more streams. Reference and historical data such as stored in your relational database can also be included in joins. Please see Section 6.13, “Joining Relational Data via SQL” for details.

Each point in time that an event arrives to one of the event streams, the two event streams are joined and output events are produced according to the where-clause.

This example joins 2 event streams. The first event stream consists of fraud warning events for which we keep the last 30 minutes (1800 seconds). The second stream is withdrawal events for which we consider the last 30 seconds. The streams are joined on account number.

```
select fraud.accountNumber as accntNum, fraud.warning as warn, withdraw.amount as amount,
       max(fraud.timestamp, withdraw.timestamp) as timestamp, 'withdrawlFraud' as desc
  from net.esper.example.atm.FraudWarningEvent.win:time(30 min) as fraud,
       net.esper.example.atm.WithdrawalEvent.win:time(30 sec) as withdraw
 where fraud.accountNumber = withdraw.accountNumber
```

Joins can also include one or more pattern statements as the next example shows:

```
select * from FraudWarningEvent.win:time(30 min) as fraud,
       pattern [every w=WithdrawalEvent -> PINChangeEvent(acct=w.acct)] as withdraw
 where fraud.accountNumber = withdraw.w.accountNumber
```

The statement above joins the last 30 minutes of fraud warnings with a pattern. The pattern consists of every withdrawal event that is followed by a PIN change event for the same account number. It joins the two event streams on account number.

6.11. Outer Joins

Esper supports left outer joins, right outer joins and full outer joins between an unlimited number of event streams. Outer joins can also join reference and historical data as explained in Section 6.13, “Joining Relational Data via SQL”.

If the outer join is a left outer join, there will be an output event for each event of the stream on the left-hand side of the clause. For example, in the left outer join shown below we will get output for each event in the stream RfidEvent, even if the event does not match any event in the event stream OrderList.

```
select * from net.esper.example.rfid.RfidEvent.win:time(30 sec) as rfid
       left outer join
       net.esper.example.rfid.OrderList.win:length(10000) as orderlist
     on rfid.itemId = orderList.itemId
```

Similarly, if the join is a Right Outer Join, then there will be an output event for each event of the stream on the right-hand side of the clause. For example, in the right outer join shown below we will get output for each event in the stream OrderList, even if the event does not match any event in the event stream RfidEvent.

```
select * from net.esper.example.rfid.RfidEvent.win:time(30 sec) as rfid
       right outer join
       net.esper.example.rfid.OrderList.win:length(10000) as orderlist
     on rfid.itemId = orderList.itemId
```

For all types of outer joins, if the join condition is not met, the select list is computed with the event properties

of the arrived event while all other event properties are considered to be null.

```
select * from net.esper.example.rfid.RfidEvent.win:time(30 sec) as rfid
       full outer join
       net.esper.example.rfid.OrderList.win:length(10000) as orderlist
     on rfid.itemId = orderList.itemId
```

The last type of outer join is a full outer join. In a full outer join, each point in time that an event arrives to one of the event streams, one or more output events are produced. In the example below, when either an RfidEvent or an OrderList event arrive, one or more output event is produced.

6.12. Subqueries

A subquery is a `select` within another statement. Esper supports subqueries in the select-clause and in the where-clause of EQL statements. Subqueries provide an alternative way to perform operations that would otherwise require complex joins. Subqueries can also make statements more readable then complex joins.

Esper supports both simple subqueries as well as correlated subqueries. In a simple subquery, the inner query is not correlated to the outer query. Here is an example simple subquery within a select-clause:

```
select assetId, (select zone from ZoneClosed.std:lastevent) as lastClosed from RFIDEvent
```

If the inner query is dependent on the outer query, we will have a correlated subquery. An example of a correlated subquery is shown below. Notice the where-clause in the inner query, where the condition involves a stream from the outer query:

```
select * from RfidEvent as RFID where 'Dock 1' =
  (select name from Zones.std:unique('zoneId') where zoneId = RFID.zoneId)
```

The example above shows a subquery in the where-clause. The statement selects RFID events in which the zone name matches a string constant based on zone id. The statement uses the view `std:unique` to guarantee that only the last event per zone id is held from processing by the subquery.

The next example is a correlated subquery within a select-clause. In this statement the select-clause retrieves the zone name by means of a subquery against the Zones set of events correlated by zone id:

```
select zoneId, (select name from Zones.std:unique('zoneId')
  where zoneId = RFID.zoneId) as name from RFIDEvent
```

Note that when a simple or correlated subquery returns multiple rows, the engine returns a `null` value as the subquery result. To limit the number of events returned by a subquery consider using one of the views `std:lastevent`, `std:unique` and `std:groupby`.

The select clause of a subquery also allows wildcard selects, which return as an event property the underlying event object of the event type as defined in the from-clause. An example:

```
select (select * from MarketData.std:lastevent()) as md
  from pattern [every timer:interval(10 sec)]
```

The output events to the statement above contain the underlying MarketData event in a property named "md". The statement populates the last MarketData event into a property named "md" every 10 seconds following the pattern definition, or populates a `null` value if no MarketData event has been encountered so far.

The following restrictions apply to subqueries:

1. The subquery stream definition must define a data window or other view to limit subquery results, reducing the number of events held for subquery execution
2. Aggregation functions cannot be used in subqueries. Instead, the insert-into clause can be used to provide aggregation results for use in subqueries
3. Subqueries can only consist of a select-clause, a from-clause and a where-clause. The group-by and having-clauses, as well as joins, outer-joins and output rate limiting are not permitted within subqueries.

Performance of your statement containing one or more subqueries principally depends on two parameters. First, if your subquery correlates one or more columns in the subquery stream with the enclosing statement's streams via equals '=', the engine automatically builds the appropriate indexes for fast row retrieval based on the key values correlated (joined). The second parameter is the number of rows found in the subquery stream and the complexity of the filter criteria (where-clause), as each row in the subquery stream must evaluate against the where-clause filter.

6.12.1. The '`exists`' keyword

The `exists` condition is considered "to be met" if the subquery returns at least one row. The `not exists` condition is considered true if the subquery returns no rows.

Let's take a look at a simple example. The following is an EQL statement that uses the `exists` condition:

```
select assetId from RFIDEvent as RFID
  where exists (select * from Asset.std:unique(assetId) where assetId = RFID.assetId)
```

This select statement will return all RFID events where there is at least one event in Assets unique by asset id with the same asset id.

6.12.2. The '`in`' keyword

The `in` subquery condition is true if the value of an expression matches one or more of the values returned by the subquery. Consequently, the `not in` condition is true if the value of an expression matches none of the values returned by the subquery.

The next statement demonstrates the use of the `in` subquery condition:

```
select assetId from RFIDEvent as RFID
  where zone in (select zone from ZoneUpdate.win:time(10 min) where status = 'closed' )
```

The above statement demonstrated the `in` subquery to select RFID events for which the zone status is in a closed state.

6.13. Joining Relational Data via SQL

This chapter outlines how reference data and historical data that are stored in a relational database can be queried via SQL within EQL statements.

Esper can join and outer join all types of event streams to stored data. In order for such data sources to become accessible to Esper, some configuration is required. The Section 2.4.5, “Relational Database Access” explains the required configuration for database access in greater detail, and includes information of configuring a query result cache.

The following restrictions currently apply:

- Only one event stream and one SQL query can be joined; Joins of two or more event streams with an SQL query are not yet supported.
- Sub-views on an SQL query are not allowed; That is, one cannot create a time or length window on an SQL query. However one can use the `insert into` syntax to make join results available to a further statement.
- Your database software must support JDBC prepared statements that provide statement meta data at compilation time. Most major databases provide this function.
- JDBC drivers must support the getMetadata feature

The next sections assume basic knowledge of SQL (Structured Query Language).

6.13.1. Joining SQL Query Results

To join an event stream against stored data, specify the `sql` keyword followed by the name of the database and a parameterized SQL query. The syntax to use in the from-clause of an EQL statement is:

```
sql:database_name [" parameterized_sql_query "]
```

The engine uses the *database_name* identifier to obtain configuration information in order to establish a database connection, as well as settings that control connection creation and removal. Please see Section 2.4.5, “Relational Database Access” to configure an engine for database access.

Following the database name is the SQL query to execute. The SQL query can contain one or more substitution parameters. The SQL query string is placed in single brackets `[` and `]`. The SQL query can be placed in either single quotes (') or double quotes ("). The SQL query grammer is passed to your database software unchanged, allowing you to write any SQL query syntax that your database understands, including stored procedure calls.

Substitution parameters in the SQL query string take the form `${`*event_property_name*`}`. The engine resolves *event_property_name* at statement execution time to the actual event property value supplied by the events in the joined event stream.

The engine determines the type of the SQL query output columns by means of the result set metadata that your database software returns for the statement. The actual query results are obtained via the `getObject` on `java.sql.ResultSet`.

The sample EQL statement below joins an event stream consisting of `CustomerCallEvent` events with the results of an SQL query against the database named `MyCustomerDB` and table `Customer`:

```
select custId, cust_name from CustomerCallEvent,
  sql:MyCustomerDB [' select cust_name from Customer where cust_id = ${custId} ']
```

The example above assumes that `CustomerCallEvent` supplies an event property named `custId`. The SQL query selects the customer name from the Customer table. The where-clause in the SQL matches the Customer table column `cust_id` with the value of `custId` in each `CustomerCallEvent` event. The engine executes the SQL query for each new `CustomerCallEvent` encountered.

If the SQL query returns no rows for a given customer id, the engine generates no output event. Else the engine generates one output event for each row returned by the SQL query. An outer join as described in the next section can be used to control whether the engine should generate output events even when the SQL query returns no rows.

The next example adds a time window of 30 seconds to the event stream `CustomerCallEvent`. It also renames the selected properties to customerName and customerId to demonstrate how the naming of columns in an SQL query can be used in the select clause in the EQL query. And the example uses explicit stream names via the `as` keyword.

```
select customerId, customerName from
  CustomerCallEvent.win:time(30 sec) as cce,
  sql:MyCustomerDB ["select cust_id as customerId, cust_name as customerName from Customer
                  where cust_id = ${cce.custId}"] as cq
```

Any window, such as the time window, generates insert stream (istream) events as events enter the window, and remove stream (rstream) events as events leave the window. The engine executes the given SQL query for each `CustomerCallEvent` in both the insert stream and the remove stream. As a performance optimization, the `istream` or `rstream` keywords in the select-clause can be used to instruct the engine to only join insert stream or remove stream events, reducing the number of SQL query executions.

6.13.2. Outer Joins With SQL Queries

You can use outer joins to join data obtained from an SQL query and control when an event is produced. Use a left outer join, such as in the next statement, if you need an output event for each event regardless of whether or not the SQL query returns rows. If the SQL query returns no rows, the join result populates null values into the selected properties.

```
select custId, custName from
  CustomerCallEvent as cce
  left outer join
  sql:MyCustomerDB ["select cust_id, cust_name as custName
                    from Customer where cust_id = ${cce.custId}"] as cq
  on cce.custId = cq.cust_id
```

The statement above always generates at least one output event for each `CustomerCallEvent`, containing all columns selected by the SQL query, even if the SQL query does not return any rows. Note the `on` expression that is required for outer joins. The `on` acts as an additional filter to rows returned by the SQL query.

6.13.3. Using Patterns to Request (Poll) Data

Pattern statements and SQL queries can also be applied together in useful ways. One such use is to poll or request data from a database at regular intervals. The next statement is an example that shows a pattern that fires every 5 seconds to query the NewOrder table for new orders:

```
insert into NewOrders
select orderId, orderAmount from
  pattern [every timer:interval(5 sec)],
  sql:MyCustomerDB ['select orderId, orderAmount from NewOrders']
```

6.13.4. JDBC Implementation Overview

The engine translates SQL queries into JDBC `java.sql.PreparedStatement` statements by replacing ${name} parameters with '?' placeholders. It obtains name and type of result columns from the compiled `PreparedStatement` meta data when the EQL statement is created.

The engine supplies parameters to the compiled statement via the `setObject` method on `PreparedStatement`. The engine uses the `getObject` method on the compiled statement `PreparedStatement` to obtain column values.

6.14. Single-row Function Reference

Single-row functions return a single value for every single result row generated by your statement. These functions can appear anywhere where expressions are allowed.

Esper allows static Java library methods as single-row functions, and also features built-in single-row functions.

Esper auto-imports the following Java library packages:

- java.lang.*
- java.math.*
- java.text.*
- java.util.*

Thus Java static library methods can be used in all expressions as shown in below example:

```
select symbol, Math.round(volume/1000)
from StockTickEvent.win:time(30 sec)
```

In general, arbitrary Java class names have to be fully qualified (e.g. java.lang.Math) but Esper provides a mechanism for user-controlled imports of classes and packages as outlined in Chapter 2, *Configuration*.

The below table outlines the built-in single-row functions available.

Table 6.2. Syntax and results of single-row functions

Single-row Function	Result
`max(expression, expression [, expression ...])`	Returns the highest numeric value among the 2 or more comma-separated expressions.
`min(expression, expression [, expression ...])`	Returns the lowest numeric value among the 2 or more comma-separated expressions.
`coalesce(expression, expression [, expression ...])`	Returns the first non-`null` value in the list, or `null` if there are no non-`null` values.

`case value`
`  when compare_value then result`
`  [when compare_value then result ...]`
`  [else result]`
`  end`
	Returns `result` where the first `value` equals `compare_value`.

`case`
`  when condition then result`
`  [when condition then result ...]`
`  [else result]`
`  end`
	Returns the `result` for the first condition that is true.

`prev(expression, event_property)`	Returns a property value of a previous event, relative to the event order within a data window
`prior(integer, event_property)`	Returns a property value of a prior event, relative to the natural order of arrival of events

6.14.1. The `Min` and `Max` Functions

The `min` and `max` function take two or more parameters that itself can be expressions. The `min` function returns the lowest numeric value among the 2 or more comma-separated expressions, while the `max` function returns the highest numeric value. The return type is the compatible aggregated type of all return values.

The next example shows the `max` function that has a `Double` return type and returns the value 1.1.

```
select max(1, 1.1, 2 * 0.5) from ...
```

The `min` function returns the lowest value. The statement below uses the function to determine the smaller of two timestamp values.

```
select symbol, min(ticks.timestamp, news.timestamp) as minT
        from StockTickEvent.win:time(30 sec) as ticks, NewsEvent.win:time(30 sec) as news
        where ticks.symbol = news.symbol
```

6.14.2. The `Coalesce` Function

The result of the `coalesce` function is the first expression in a list of expressions that returns a non-null value. The return type is the compatible aggregated type of all return values.

This example returns a String-typed result of value 'foo'.

```
select coalesce(null, 'foo') from ...
```

6.14.3. The `Case` Control Flow Function

The `case` control flow function has two versions. The first version takes a value and a list of compare values to compare against, and returns the result where the first value equals the compare value. The second version takes a list of conditions and returns the result for the first condition that is true.

The return type of a `case` expression is the compatible aggregated type of all return values.

The example below shows the first version of a `case` statement. It has a `String` return type and returns the value 'one'.

```
select case 1 when 1 then 'one' when 2 then 'two' else 'more' end from ...
```

The second version of the `case` function takes a list of conditions. The next example has a `Boolean` return type and returns the boolean value true.

```
select case when 1>0 then true else false end from ...
```

6.14.4. The `Previous` Function

The `prev` function returns the property value of a previous event. The first parameter denotes the i-th previous event in the order established by the data window. The second parameter is a property name for which the function returns the value for the previous event.

This example selects the value of the `price` property of the 2nd-previous event from the current Trade event.

```
select prev(2, price) from Trade.win:length(10)
```

Since the prev function takes the order established by the data window into account, the function works well with sorted windows. In the following example the statement selects the symbol of the 3 Trade events that had the largest, second-largest and third-largest volume.

```
select prev(0, symbol), prev(1, symbol), prev(2, symbol)
  from Trade.ext:sort(volume, true, 10)
```

The i-th previous event parameter can also be an expression returning an Integer-type value. The next statement joins the Trade data window with an RankSelectionEvent event that provides a rank property used to look up a certain position in the sorted Trade data window:

```
select prev(rank, symbol) from Trade.ext:sort(volume, true, 10), RankSelectionEvent
```

And the expression count(*) - 1 allows us to select the oldest event in the length window:

```
select prev(count(*) - 1, price) from Trade.win:length(100)
```

The prev function returns a null value if the data window does not currently hold the i-th previous event. The example below illustrates this using a time batch window. Here the prev function returns a null value for any events in which the previous event is not in the same batch of events. Note that the prior function as discussed below can be used if a null value is not the desired result.

```
select prev(1, symbol) from Trade.win:time_batch(1 min)
```

Previous Event per Group

The combination of prev function and group-by view returns the property value for a previous event in the given group.

Let's look at an example. Assume we want to obtain the price of the previous event of the same symbol as the current event.

The statement that follows solves this problem. It declares a group-by view grouping on the symbol property and a time window of 1 minute. As a result, when the engine encounters a new symbol value that it hasn't seen before, it creates a new time window specifically to hold events for that symbol. Consequently, the previous function returns the previous event within the respective time window for that event's symbol value.

```
select prev(1, price) as prevPrice from Trade.std:groupby('symbol').win:time(1 min)
```

In a second example, assume we need to return, for each event, the current top price per symbol. We can use the prev to obtain the highest price from a sorted data window, and use the group-by view to group by symbol:

```
select prev(0, price) as topPricePerSymbol
from Trade.std:groupby('symbol').ext:sort('price', false, 1)
```

Restrictions

The following restrictions apply to the prev functions and its results:

- The function always returns a null value for remove stream (old data) events
- The function requires a data window view, or a group-by and data window view, without any additional sub-views. Data window views are: length window, time and time batch window and sorted window

Comparison to the prior Function

The prev function is similar to the prior function. The key differences between the two functions are as follows:

- The prev function returns previous events in the order provided by the data window, while the prior function returns prior events in the order of arrival as posted by a stream's declared views.
- The prev function requires a data window view while the prior function does not have any view requirements.
- The prev function returns the previous event grouped by a criteria by combining the std:groupby view and a data window. The prior function returns prior events posted by the last view regardless of data window grouping.
- The prev function returns a null value for remove stream events, i.e. for events leaving a data window. The prior function does not have this restriction.

6.14.5. The Prior Function

The prior function returns the property value of a prior event. The first parameter is an integer value that denotes the i-th prior event in the natural order of arrival. The second parameter is a property name for which the function returns the value for the prior event.

This example selects the value of the price property of the 2nd-prior event to the current Trade event.

```
select prior(2, price) from Trade
```

The prior function can be used on any event stream or view and does not have any specific view requirements. The function operates on the order of arrival of events by the event stream or view that provides the events.

The next statement uses a time batch window to compute an average volume for 1 minute of Trade events, posting results every minute. The select-clause employs the prior function to select the current average and the average before the current average:

```
select average, prior(1, average)
    from TradeAverages.win:time_batch(1 min).stat:uni('volume')
```

6.15. Operator Reference

Esper arithmatic and logical operator precedence follows Java standard arithmatic and logical operator precedence.

6.15.1. Arithmatic Operators

The below table outlines the arithmatic operators available.

Table 6.3. Syntax and results of arithmatic operators

Operator	Description
+, -	As unary operators they denote a positive or negative expression. As binary operators they add or subtract.

Operator	Description
*, /	Multiplication and division are binary operators.
%	Modulo binary operator.

6.15.2. Logical And Comparsion Operators

The below table outlines the logical and comparison operators available.

Table 6.4. Syntax and results of logical and comparison operators

Operator	Description
NOT	Returns true if the following condition is false, returns false if it is true.
OR	Returns true if either component condition is true, returns false if both are false.
AND	Returns true if both component conditions are true, returns false if either is false.
=, !=, <, > <=, >=,	Comparison.

6.15.3. Concatenation Operators

The below table outlines the concatenation operators available.

Table 6.5. Syntax and results of concatenation operators

Operator	Description
||	Concatenates character strings

6.15.4. Binary Operators

The below table outlines the binary operators available.

Table 6.6. Syntax and results of binary operators

Operator	Description
&	Bitwise AND if both operands are numbers; conditional AND if both operands are boolean
|	Bitwise OR if both operands are numbers; conditional OR if both operands are boolean
^	Bitwise exclusive OR (XOR)

6.15.5. Array Definition Operator

The `{` and `}` curly braces are array definition operators following the Java array initialization syntax. Arrays can be useful to pass to user-defined functions or to select array data in a select clause.

Array definitions consist of zero or more expressions within curly braces. Any type of expression is allowed within array definitions including constants, arithmatic expressions or event properties. This is the syntax of an array definition:

```
{ [expression [,expression...]] }
```

Consider the next statement that returns an event property named `actions`. The engine populates the `actions` property as an array of `java.lang.String` values with a length of 2 elements. The first element of the array contains the `observation` property value and the second element the `command` property value of `RFIDEvent` events.

```
select {observation, command} as actions from RFIDEvent
```

The engine determines the array type based on the types returned by the expressions in the array definiton. For example, if all expressions in the array definition return integer values then the type of the array is `java.lang.Integer[]`. If the types returned by all expressions are compatible number types, such as integer and double values, the engine coerces the array element values and returns a suitable type, `java.lang.Double[]` in this example. The type of the array returned is `Object[]` if the types of expressions cannot be coerced or return object values. Null values can also be used in an array definition.

Arrays can come in handy for use as parameters to user-defined functions:

```
select * from RFIDEvent where Filter.myFilter(zone, {1,2,3})
```

6.15.6. The '`in`' Keyword

The `in` keyword determines if a given value matches any value in a list. The syntax of the keyword is:

```
test_expression [not] in (expression [,expression...] )
```

The *test_expression* is any valid expression. The keyword is followed by a list of expressions to test for a match. The optional `not` keyword specifies that the result of the predicate be negated.

The result of an `in` expression is of type `Boolean`. If the value of *test_expression* is equal to any expression

from the comma-separated list, the result value is `true`. Otherwise, the result value is `false`. All expressions must be of the same type as or a compatible type to *test_expression*.

The next example shows how the `in` keyword can be applied to select certain command types of RFID events:

```
select * from RFIDEvent where command in ('OBSERVATION', 'SIGNAL')
```

The statement is equivalent to:

```
select * from RFIDEvent where command = 'OBSERVATION' or command = 'SIGNAL'
```

6.15.7. The '`between`' Keyword

The `between` keyword specifies a range to test. The syntax of the keyword is:

```
test_expression [not] between begin_expression and end_expression
```

The *test_expression* is any valid expression and is the expression to test for in the range defined by *begin_expression* and *end_expression*. The `not` keyword specifies that the result of the predicate be negated.

The result of a `between` expression is of type `Boolean`. If the value of *test_expression* is greater then or equal to the value of *begin_expression* and less than or equal to the value of *end_expression*, the result is `true`.

The next example shows how the `between` keyword can be used to select events with a price between 55 and 60 (inclusive).

```
select * from StockTickEvent where price between 55 and 60
```

The equivalent expression without `between` is:

```
select * from StockTickEvent where price >= 55 and price <= 60
```

And also equivalent to:

```
select * from StockTickEvent where price between 60 and 55
```

6.15.8. The '`like`' Keyword

The `like` keyword provides standard SQL pattern matching. SQL pattern matching allows you to use `'_'` to match any single character and `'%'` to match an arbitrary number of characters (including zero characters). In Esper, SQL patterns are case-sensitive by default. The syntax of `like` is:

```
test_expression [not] like pattern_expression [escape string_literal]
```

The *test_expression* is any valid expression yielding a String-type or a numeric result. The optional `not` keyword specifies that the result of the predicate be negated. The `like` keyword is followed by any valid standard SQL *pattern_expression* yielding a String-typed result. The optional `escape` keyword signals the escape character to escape `'_'` and `'%'` values in the pattern.

The result of a `like` expression is of type `Boolean`. If the value of *test_expression* matches the *pattern_expression*, the result value is `true`. Otherwise, the result value is `false`.

An example for the `like` keyword is below.

```
select * from PersonLocationEvent where name like '%Jack%'
```

The escape character can be defined as follows. In this example the where-clause matches events where the suffix property is a single `'_'` character.

```
select * from PersonLocationEvent where suffix like '!_' escape '!'
```

6.15.9. The '`regexp`' Keyword

The `regexp` keyword is a form of pattern matching based on regular expressions implemented through the Java `java.util.regex` package. The syntax of `regexp` is:

```
test_expression [not] regexp pattern_expression
```

The *test_expression* is any valid expression yielding a String-type or a numeric result. The optional `not` keyword specifies that the result of the predicate be negated. The `regexp` keyword is followed by any valid regular expression *pattern_expression* yielding a String-typed result.

The result of a `regexp` expression is of type `Boolean`. If the value of *test_expression* matches the regular expression *pattern_expression*, the result value is `true`. Otherwise, the result value is `false`.

An example for the `regexp` keyword is below.

```
select * from PersonLocationEvent where name regexp '*Jack*'
```

6.16. Built-in views

This chapter outlines the views that are built into Esper. All views can be arbitrarily combined as many of the examples below show. The section on Chapter 4, *Understanding the Output Model* provides additional information on the relationship of views, filtering and aggregation.

6.16.1. Window views

Length window (`win:length`)

Creates a moving window extending the specified number of elements into the past. The view takes a single numeric parameter that defines the window size:

```
win:length(size)
```

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

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

The next example keeps a length window of 10 events of stock trade events, with a separate window for each symbol. The statistics on price is calculated only for the last 10 events for each symbol.

```
select * from StockTickEvent.std:groupby('symbol').win:length(10).stat:uni('price')
```

Length window batch (`win:length_batch`)

This window view buffers events and releases them when a given minimum number of events has been collected. The view takes the number of events to batch as a parameter:

```
win:length_batch(size)
```

The next statement buffers events until a minimum of 5 events have collected. Listeners to updates posted by this view receive updated information only when 5 or more events have collected.

```
select * from StockTickEvent.win:length_batch(5)
```

Time window (`win:time`)

Creates a moving time window extending from the specified time interval into the past based on the system time. This view takes a time period (see Section 6.2.1, “Specifying Time Periods”) or a number of seconds as a parameter:

```
win:time(time period)
```

```
win:time(number of seconds)
```

For the IBM stock tick events in the last 1 second, calculate statistics on price.

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

The same statement rewritten to use a parameter supplying number-of-seconds is:

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

The following time windows are equivalent specifications:

```
win:time(2 minutes 5 seconds)
win:time(125 sec)
win:time(125)
```

Externally-timed window (`win:ext_timed`)

Similar to the time window, this view is a moving time window extending from the specified time interval into the past, but based on the millisecond time value supplied by an event property. The view takes two parameters: the name of the event property to return the long-typed timestamp value, and a time period or a number of seconds:

```
win:time(timestamp_property_name, time_period)
```

```
win:time(timestamp_property_name, number_of_seconds)
```

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

```
select * from StockTickEvent.win:ext_timed('timestamp', 10 seconds)
```

Time window batch (`win:time_batch`)

This window view buffers events and releases them every specified time interval in one update. The view takes a time period or a number of seconds as a parameter.

```
win:time_batch(time_period)
```

```
win:time_batch(number_of_seconds)
```

The below example batches 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.

```
select * from StockTickEvent.win:time_batch(5 sec)
```

6.16.2. Standard view set

Unique (`std:unique`)

The `unique` view is a view that includes only the most recent among events having the same value for the specified field:

```
std:unique(event_property_name)
```

The view acts as a length window of size 1 for each distinct value of the event property. It thus posts as old events the prior event of the same property value, if any.

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

```
select * from StockTickEvent.std:unique('symbol')
```

Group By (`std:groupby`)

This view groups events into sub-views by the value of the specified field. The view takes a single property name to supply the group-by values, or a list of property names as the synopsis shows:

```
std:groupby(property_name)
```

```
std:groupby({property_name [, property_name ...] })
```

This example calculates statistics on price separately for each symbol.

```
select * from StockTickEvent.std:groupby('symbol').stat:uni('price')
```

The group-by view can also take multiple fields to group by. This example calculates statistics on price for each symbol and feed.

```
select * from StockTickEvent.std:groupby({'symbol', 'feed'}).stat:uni('price')
```

The order in which the group-by view appears within sub-views of a stream controls the data the engine derives from events for each group. The next 2 statements demonstrate this using a length window.

This example keeps a length window of 10 events of stock trade events, with a separate length window for each symbol. The engine calculates statistics on price for the last 10 events for each symbol. During runtime, the engine actually allocates a separate length window for each new symbol arriving.

```
select * from StockTickEvent.std:groupby('symbol').win:length(10).stat:uni('price')
```

By putting the group-by view in position after the length window, we can change the semantics of the query. The query now returns the statistics on price per symbol for only the last 10 events across all symbols. Here the engine allocates only one length window for all events.

```
select * from StockTickEvent.win:length(10).std:groupby('symbol').stat:uni('price')
```

We have learned that by placing the group-by view before other views, these other views become part of the grouped set of views. The engine dynamically allocates a new view instance for each subview, every time it encounters a new group key such as a new value for symbol. Therefore, in `std:groupby('symbol').win:length(10)` the engine allocates a new length window for each distinct symbol. However in `win:length(10).std:groupby('symbol')` the engine maintains a single length window.

Multiple group-by views can also be used in the same statement. The statement below groups by symbol and feed. As the statement declares the time window after the group-by view for symbols, the engine allocates a new time window per symbol however reports statistics on price per symbol and feed. The query results are statistics on price per symbol and feed for the last 1 minute of events per symbol (and not per feed).

```
select * from StockTickEvent.std:groupby('symbol').win:time(1 minute)
    .std:groupby('feed').stat:uni('price')
```

Last, we consider the permutation where the time window is declared after the group-by. Here, the query results are statistics on price per symbol and feed for the last 1 minute of events per symbol and feed.

```
select * from StockTickEvent.std:groupby({'symbol', 'feed'})
    .win:time(1 minute).stat:uni('price')
```

Size (`std:size`)

This view simply posts the number of events received from a stream or view. The synopsis is simply:

```
std:size()
```

The view posts a single long-typed property named `size`. The view posts the prior size as old data, and the current size as new data to update listeners of the view. Via the `iterator` method of the statement the size value can also be polled (read).

When combined with a data window view, the size view reports the current and prior number of events in the data window. This example reports the number of tick events within the last 1 minute:

```
select size from StockTickEvent.win:time(1 min).std:size()
```

The size view is also useful in conjunction with a group-by view to count the number of events per group. The EQL below returns the number of events per symbol.

```
select size from StockTickEvent.std:groupby('symbol').std:size()
```

When used without a data window, the view simply counts the number of events:

```
select size from StockTickEvent.std:size()
```

All views can be used with pattern statements as well. The next EQL snippet shows a pattern where we look for tick events followed by trade events for the same symbol. The size view counts the number of occurances of the pattern.

```
select size from pattern[every s=StockTickEvent -> TradeEvent(symbol=s.symbol)].std:size()
```

Last (`std:lastevent`)

This view exposes the last element of its parent view:

```
std:lastevent()
```

The view acts as a length window of size 1. It thus posts as old events the prior event in the stream, if any.

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

```
select * from StockTickEvent(symbol='IBM').stat:uni('price').std:lastevent()
```

6.16.3. Statistics views

Univariate statistics (`stat:uni`)

This view calculates univariate statistics on an event property. The view takes a single event property name as a parameter. The event property must be of numeric type:

```
stat:uni(event_property_name)
```

Table 6.7. 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 selects the standard deviation on price for stock tick events for the last 10 events.

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

Regression (`stat:linest`)

This view calculates regression on two event properties. The view takes two event property names as parameters. The event properties must be of numeric type:

```
stat:linest(event_property_name_1, event_property_name_2)
```

Table 6.8. 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.

```
select slope, YIntercept from StockTickEvent.win:time(10 seconds).stat:linest('price', 'offer')
```

Correlation (stat:correl)

This view calculates the correlation value on two event properties. The view takes two event property names as parameters. The event properties must be of numeric type:

```
stat:correl(event_property_name_1, event_property_name_2)
```

Table 6.9. Correlation derived properties

Property Name	Description
correlation	Correlation between two event properties

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

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

Weighted average (stat:weighted_avg)

This view returns the weighted average given a weight field and a field to compute the average for. The view takes two event property names as parameters. The event properties must be of numeric type:

```
stat:weighted_avg(event_property_name_field, event_property_name_weight)
```

Table 6.10. Weighted average derived properties

Property Name	Description
average	Weighted average

A statement that derives the volume-weighted average price for the last 3 seconds:

```
select average
from StockTickEvent(symbol='IBM').win:time(3 seconds).stat:weighted_avg('price', 'volume')
```

Multi-dimensional statistics (stat:cube)

This view works similar to the std:groupby 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.

```
stat:cube(values_to_derive, property_name_datapoint, property_name_column)
```

```
stat:cube(values_to_derive, property_name_datapoint, property_name_column,
    property_name_row)
```

```
stat:cube(values_to_derive, property_name_datapoint, property_name_column,
    property_name_row, property_name_page)
```

Table 6.11. Multi-dim derived properties

Property Name	Description
`cube`	The cube following the `net.esper.view.stat.olap.Cube` interface

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

```
select cube from ServiceMeasurement.stat:cube({‘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.

```
select * from ServiceMeasurement.win:length(30000).stat:cube({‘average’},
       'latency', 'customer', 'service', 'status')
```

6.16.4. Extension View Set

Sorted Window View (`ext:sort`)

This view sorts by values of the specified event properties and keeps only the top events up to the given size.

The syntax to sort on a single event property is as follows.

```
std:sort(property_name, is_descending, size)
```

To sort on a multiple event properties the syntax is as follows.

```
sort( { property_name, is_descending [ , property_name, is_descending ...] }, size)
```

The view below sorts on price descending keeping the lowest 10 prices and reporting statistics on price.

```
select * from StockTickEvent.ext:sort('price', false, 10).stat:uni('price')
```

The following example sorts events first by price in descending order, and then by symbol name in ascending (alphabetical) order, keeping only the 10 events with the highest price (with ties resolved by alphabetical order of symbol).

```
select * from StockTickEvent.ext:sort({'price', true, 'symbol', false}, 10)
```

6.17. User-Defined Functions

A user-defined function can be invoked anywhere as an expression itself or within an expresson. The function must simply be a public static method that the classloader can resolve at statement creation time. The engine resolves the function reference at statement creation time and verifies parameter types.

The example below assumes a class `MyClass` that exposes a public static method `myFunction` accepting 2 parameters, and returing a numeric type such as `double`.

```
select 3 * MyClass.myFunction(price, volume) as myValue
from StockTick.win:time(30 sec)
```

User-defined functions also take array parameters as this example shows. The section on Section 6.15.5, “Array Definition Operator” outlines in more detail the types of arrays produced.

```
select * from RFIDEvent where com.mycompany.rfid.MyChecker.isInZone(zone, {10, 20, 30})
```

Chapter 7. Event Pattern Reference

7.1. 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 sub-expression repetition: `every`
2. Logical operators: `and`, `or`, `not`
3. Temporal operators that operate on event order: `->` (followed-by)
4. Guards are where-conditions that control the lifecycle of sub-expressions. Examples are `timer:within`.
5. Observers observe time events as well as other events. Examples are `timer:interval` and `timer:at`.

7.2. How to use Patterns

7.2.1. Pattern Syntax

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` event in which the `success` property is false. Either one or the other condition must be true for this pattern to match.

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

In the example above, 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 7.3, “Pattern Filter Expressions”.

The example above assigned the tags `spike` and `error` to the events in the pattern. The tags are important since the engine only places tagged events into the output event(s) that a pattern generates, and that the engine supplies to listeners of the pattern statement. The tags can further be selected in the select-clause of an EQL statement as discussed in Section 6.4.2, “Pattern-based event streams”.

Pattern statements are created via the `EPAdministrator` interface. The `EPAdministrator` interface allows to create pattern statements in two ways: Pattern statements that want to make use of the EQL `select` clause or any other EQL constructs use the `createEQL` method to create a statement that specifies one or more pattern expressions. EQL statements that use patterns are described in more detail in Section 6.4.2, “Pattern-based event streams”. Use the syntax as shown in below example.

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

String eventName = ServiceMeasurement.class.getName();

EPStatement myTrigger = admin.createEQL("select * from pattern [" +
  "every (spike=" + eventName + "(latency>20000) or error=" + eventName + "(success=false))]");
```

```
```

Pattern statements that do not need to make use of the EQL `select` clause or any other EQL constructs can use the `createPattern` method, as in below example.

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

7.2.2. Subscribing to Pattern Events

When a pattern fires it publishes one or more events to any listeners to the pattern statement. The listener interface 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 occurred
    ... // 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`.

7.2.3. 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
}
```

7.3. Pattern Filter Expressions

The simplest form of filter is a filter for events of a given type without any conditions on the event property

values. This filter matches any event of that type regardless of the event's properties. The example below is such a filter. Note that this event pattern would stop firing as soon as the first RfidEvent is encountered.

```
com.mypackage.myevents.RfidEvent
```

To make the event pattern fire for every RfidEvent and not just the first event, use the `every` keyword.

```
every com.mypackage.myevents.RfidEvent
```

The example above specifies the fully-qualified Java class name as the event type. Via configuration, the event pattern above can be simplified by using the alias that has been defined for the event type.

```
every RfidEvent
```

Interfaces and superclasses are also supported as event types. In the below example `IRfidReadable` is an interface class, and the statement matches any event that implements this interface:

```
every org.myorg.rfid.IRfidReadable
```

The filtering criteria to filter for events with certain event property values are placed within parenthesis after the event type name:

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

All expressions can be used in filters, including static method invocations that return a boolean value:

```
RfidEvent(MyRFIDLib.isInRange(x, y) or (x<0 and y < 0))
```

Filter expressions can be separated via a single comma ',''. The comma represents a logical AND between expressions:

```
RfidEvent(zone=1, category=10)
...is equivalent to...
RfidEvent(zone=1 and category=10)
```

The following set of operators are highly optimized through indexing and are the preferred means of filtering high-volume event streams:

- equals `=`
- not equals `!=`
- comparison operators `< , > , >=, <=`
- ranges
  - use the `between` keyword for a closed range where both endpoints are included
  - use the `in` keyword and round `()` or square brackets `[]` to control how endpoints are included
  - for inverted ranges use the `not` keyword and the `between` or `in` keywords
- list-of-values checks using the `in` keyword or the `not in` keywords followed by a comma-separated list of values

At compile time as well as at run time, the engine scans new filter expressions for sub-expressions that can be indexed. Indexing filter values to match event properties of incoming events enables the engine to match incoming events faster. The above list of operators represents the set of operators that the engine can best convert into indexes. The use of comma or logical `and` in filter expressions does not impact optimizations by the engine.

For more information on filters please see Section 6.4.1, “Filter-based event streams”.

Filter criteria can also refer to events matching prior named events in the same expression. Below pattern is an example in which the pattern matches once for every RfidEvent that is preceded by an RfidEvent with the same asset id.

```
every A=RfidEvent -> B=RfidEvent(assetId=A.assetId)
```

The syntax shown above allows filter criteria to reference prior results by specifying the event name tag of the prior event, and the event property name. This syntax can be used in all filter operators or expressions including ranges and the `in` set-of-values check:

```
every A=RfidEvent ->
  B=RfidEvent(MyLib.isInRadius(A.x, A.y, x, y) and zone in (1, A.zone))
```

7.4. Pattern Operators

7.4.1. Every

The `every` operator indicates that the pattern sub-expression should restart when the sub-expression qualified by the `every` keyword evaluates to true or false. Without the `every` operator the pattern sub-expression stops when the pattern sub-expression evaluates to true or false.

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

Every time a pattern sub-expression within an `every` operator turns true the engine starts a new active sub-expression looking for more event(s) or timing conditions that match the pattern sub-expression. If the `every` operator is not specified for a sub-expression, the sub-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
```

Let's 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 7.1. '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}

Example	Description
`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.

Let's consider the `every` operator in conjunction with a sub-expression that matches 3 events that follow each other:

```
every (A -> B -> C)
```

The pattern first looks for event A. When event A arrives, it looks for event B. After event B arrives, the pattern looks for event C. Finally when event C arrives the pattern fires. The engine then starts looking for event A again.

Assume that between event B and event C a second event A_2 arrives. The pattern would ignore the A_2 entirely since it's then looking for event C. As observed in the prior example, the `every` operator restarts the sub-expression `A -> B -> C` only when the sub-expression fires.

In the next statement the `every` operator applies only to the A event, not the whole sub-expression:

```
every A -> B -> C
```

This pattern now matches for any event A that is followed by an event B and then event C, regardless of when the event A arrives. Oftentimes this can be practical in combination with the `and not` syntax and the `timer:within` syntax as the next example shows.

This example looks at temperature sensor events named Sample. The pattern detects when 3 sensor events indicate a temperature of more then 50 degrees uninterrupted within 90 seconds of the first event, considering events for the same sensor only.

```
every sample=Sample(temp > 50) ->
```

```
  ( (Sample(sensor=sample.sensor, temp > 50) and not Sample(sensor=sample.sensor, temp <= 50))
      ->
    (Sample(sensor=sample.sensor, temp > 50) and not Sample(sensor=sample.sensor, temp <= 50))
  ) where timer:within(90 seconds))
```

The pattern starts a new sub-expression in the round braces after the first followed-by operator for each time a sensor indicated more then 50 degrees. Each sub-expression then lives a maximum of 90 seconds. Each sub-expression ends if a temperature of 50 degress or less is encountered for the same sensor. Only if 3 temperature events in a row indicate more then 50 degrees, and within 90 seconds of the first event, and for the same sensor, does this pattern fire.

7.4.2. 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)
```

Note that in an `and` pattern expression it is not possible to correlate events based on event property values. For example, this is an invalid pattern:

```
// This is NOT valid
a=A and B(id = a.id)
```

The above expression is invalid as it relies on the order of arrival of events, however in an `and` expression the order of events is not specified and events fulfill an `and` condition in any order. The above expression can be changed to use the followed-by operator:

```
// This is valid
a=A -> B(id = a.id)
// another example using 'and'...
a=A -> (B(id = a.id) and C(id = a.id))
```

7.4.3. 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)
```

7.4.4. 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
```

7.4.5. 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
```

This is a pattern that fires when 2 status events indicating an error occur one after the other.

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

7.5. Pattern Guards

Guards are where-conditions that control the lifecycle of sub-expressions. Custom guard functions can also be used. The section Chapter 8, *Extension and Plug-in* outlines guard plug-in development in greater detail.

Take as an example the following pattern expression:

```
MyEvent where timer.within(10 sec)
```

In this pattern the `timer:within` guard controls the sub-expression that is looking for MyEvent events. The guard terminates the sub-expression looking for MyEvent events after 10 seconds after start of the pattern. Thus the pattern alerts only once when the first MyEvent event arrives within 10 seconds after start of the pattern.

The `every` keyword requires additional discussion since it also controls sub-expression lifecycle. Let's add the `every` keyword to the example pattern:

```
every MyEvent where timer.within(10 sec)
```

The difference to the pattern without `every` is that each MyEvent event that arrives now starts a new sub-expression, including a new guard, looking for a further MyEvent event. The result is that, when a MyEvent arrives within 10 seconds after pattern start, the pattern execution will look for the next MyEvent event to arrive within 10 seconds after the previous one.

By placing parentheses around the `every` keyword and its sub-expression, we can have the `every` under the control of the guard:

```
(every MyEvent) where timer.within(10 sec)
```

In the pattern above, the guard terminates the sub-expression looking for all MyEvent events after 10 seconds after start of the pattern. This pattern alerts for all MyEvent events arriving within 10 seconds after pattern start, and then stops.

7.5.1. 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. The `timer:within` guard takes a time period (see Section 6.2.1, “Specifying Time Periods”) or a number of seconds as a parameter.

This pattern fires if an A event arrives within 5 seconds after statement creation.

```
A where timer:within (5 seconds)
```

This pattern fires for all A events that arrive within 5 seconds. After 5 seconds, this pattern stops matching even if more A events arrive.

```
(every A) where timer:within (5 seconds)
```

This pattern is similar to the first pattern but here every time A arrives within 5 seconds, the pattern begins looking for A for another 5 seconds. As long as A events arrive within 5 seconds after the last A, the pattern does not stop matching.

```
every (A where timer:within (5 sec))
```

This pattern matches for any one A or B event in the next 5 seconds.

```
( A or B ) where timer:within (5 sec)
```

This pattern matches for any 2 errors that happen 10 seconds within each other.

```
every (StatusEvent(status='ERROR') -> StatusEvent(status='ERROR') where timer:within (10 sec))
```

The following guards are equivalent:

```
timer:within(2 minutes 5 seconds)
timer:within(125 sec)
timer:within(125)
```

7.6. Pattern Observers

Observers observe time-based events for which the thread-of-control originates by the engine timer thread. Custom observers can also be developed that observe timer events or other engine-external events. The section Chapter 8, *Extension and Plug-in* outlines observer plug-in development in greater detail.

7.6.1. timer:interval

The `timer:interval` observer waits for the defined time before the truth value of the observer turns true. The observer takes a time period (see Section 6.2.1, “Specifying Time Periods”) or a number of seconds as a parameter.

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

```
A -> timer:interval(10 seconds)
```

The pattern below fires every 20 seconds.

```
every timer:interval(20 sec)
```

The next example pattern fires for every event A that is not followed by an event B within 60 seconds after event A arrived. B must have the same "id" property value as A.

```
every a=A -> (timer:interval(60 sec) and not B(id=a.id))
```

7.6.2. 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 x_{th} 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 `timer:at` 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], *, *)
```

The below table outlines the fields, valid values and keywords available for each field:

Table 7.2. Properties offered by sample statement aggregating price

Field Name	Mandatory?	Allowed Values	Additional Keywords
Minutes	yes	0 - 59	
Hours	yes	0 - 23	
Days Of Month	yes	1 - 31	last, weekday, lastweekday
Months	yes	1 - 12	
Days Of Week	yes	0 (Sunday) - 6 (Saturday)	last
Seconds	no	0 - 59	

The keyword `last` used in the days-of-month field means the last day of the month (current month). To specify

the last day of another month, a value for the month field has to be provided. For example: `timer:at(*, *, last,2,*)` is the last day of February.

The `last` keyword in the day-of-week field by itself simply means Saturday. If used in the day-of-week field after another value, it means "the last xxx day of the month" - for example "5 last" means "the last friday of the month". So the last Friday of the current month will be: `timer:at(*, *, *, *, 5 last)`. And the last Friday of June: `timer:at(*, *, *, 6, 5 last)`.

The keyword `weekday` is used to specify the weekday (Monday-Friday) nearest the given day. Variant could include month like in: `timer:at(*, *, 30 weekday, 9, *)` which is Friday September 28th (no jump over month).

The keyword `lastweekday` is a combination of two parameters, the `last` and the `weekday` keywords. A typical example could be: `timer:at(*, *, *, lastweekday, 9, *)` which will define Friday September 28th (example year is 2007).

Chapter 8. Extension and Plug-in

8.1. Overview

Esper can currently be extended by these means:

- User-defined functions - these can be used anywhere where expressions are allowed, please see Section 6.17, “User-Defined Functions”
- Custom-developed Plug-in Views

8.2. Custom View Implementation

Views in Esper are used to derive information from an event stream, and to represent data windows onto an event stream. This chapter describes how to plug-in a new, custom view.

The following steps are required to develop and use a custom view with Esper.

1. Implement a view factory class. View factories are classes that accept and check view parameters and instantiate the appropriate view class.
2. Implement a view class. A view class commonly represents a data window or derives new information from a stream.
3. Configure the view factory class supplying a view namespace and name in the engine configuration file.

The example view factory and view class that are used in this chapter can be found in the test source folder in the package `net.esper.regression.client` by the name `MyTrendSpotterViewFactory` and `MyTrendSpotterView`.

Views can make use of the following engine services available via `StatementServiceContext`:

- The `SchedulingService` interface allows views to schedule timer callbacks to a view
- The `EventAdapterService` interface allows views to create new event types and event instances of a given type.
- The `StatementStopService` interface allows view to register a callback that the engine invokes to indicate that the view's statement has been stopped

Note that custom views may use engine services and APIs that can be subject to change between major releases. The engine services discussed above and view APIs are considered part of the engine internal public API and are stable. Any changes to such APIs are disclosed through the release change logs and history. Please also consider contributing your custom view to the Esper project team by submitting the view code through the mailing list or via a JIRA issue.

8.2.1. Implementing a View Factory

A view factory class is responsible for the following functions:

- Accept zero, one or more view parameters. Validate and parse the parameters as required.
- Validate that the parameterized view is compatible with its parent view. For example, validate that field names are valid in the event type of the parent view.
- Instantiate the actual view class.
- Provide information about the event type of events posted by the view.

View factory classes simply subclass `net.esper.view.ViewFactorySupport`:

```
public class MyTrendSpotterViewFactory extends ViewFactorySupport { ...
```

Your view factory class must implement the `setViewParameters` method to accept and parse view parameters. The next code snippet shows an implementation of this method. The code obtains a single field name parameter from the parameter list passed to the method:

```
public class MyTrendSpotterViewFactory extends ViewFactorySupport {
  private String fieldName;
  private EventType eventType;

  public void setViewParameters(ViewFactoryContext viewFactoryContext,
                          List<Object> viewParameters) throws ViewParameterException
  {
    String errorMessage = "'Trend spotter' view require a single field name as a parameter";
    if (viewParameters.size() != 1) {
      throw new ViewParameterException(errorMessage);
    }

    if (!(viewParameters.get(0) instanceof String)) {
      throw new ViewParameterException(errorMessage);
    }

    fieldName = (String) viewParameters.get(0);
  }
  ...
```

After the engine supplied view parameters to the factory, the engine will ask the view to attach to its parent view and validate any field name parameters against the parent view's event type. If the view will be generating events of a different type then the events generated by the parent view, then the view factory can create the new event type in this method:

```
public void attach(EventType parentEventType,
                StatementServiceContext statementServiceContext,
                ViewFactory optionalParentFactory,
                List<ViewFactory> parentViewFactories)
                    throws ViewAttachException {
  String result = PropertyCheckHelper.checkNumeric(parentEventType, fieldName);
  if (result != null) {
    throw new ViewAttachException(result);
  }

  // create new event type
  Map<String, Class> eventTypeMap = new HashMap<String, Class>();
  eventTypeMap.put(PROPERTY_NAME, Long.class);
  eventType = statementServiceContext.getEventAdapterService().
                        createAnonymousMapType(eventTypeMap);
}
```

Finally, the engine asks the view factory to create a view instance:

```
public View makeView(StatementServiceContext statementServiceContext) {
  return new MyTrendSpotterView(statementServiceContext, fieldName);
}
```

8.2.2. Implementing a View

A view class is responsible for:

- The `setParent` method informs the view of the parent view's event type

- The `update` method receives insert streams and remove stream events from its parent view
- The `iterator` method supplies an (optional) iterator to allow an application to pull or request results from an `EPStatement`
- The `cloneView` method must make a configured copy of the view to enable the view to work in a grouping context together with a `std:groupby` parent view

View classes simply subclass `net.esper.view.ViewSupport`:

```
public class MyTrendSpotterView extends ViewSupport { ...
```

The view class must implement the `setParent(Viewable parent)` method. This is an opportunity for the view to initialize and obtain a fast event property getter for later use to obtain event property values. The next code snippet shows an implementation of this method:

```
public void setParent(Viewable parent) {
  super.setParent(parent);
  if (parent != null)  {
    fieldGetter = parent.getEventType().getGetter(fieldName);
  }
}
```

Your `update` method will be processing incoming (insert stream) and outgoing (remove stream) events, as well as providing incoming and outgoing events to child views. The convention required of your update method implementation is that the view releases any insert stream events which the view generates as semantically-equal remove stream events at a later time. A sample `update` method implementation that computes a number of events in an upward trend is shown below:

```
public final void update(EventBean[] newData, EventBean[] oldData) {
  EventBean oldDataPost = populateMap(trendcount);

  // add data points
  if (newData != null) {
    for (int i = 0; i < newData.length; i++) {
      double dataPoint = ((Number) fieldGetter.get(newData[i])).doubleValue();

      if (lastDataPoint == null) {
        trendcount = 1L;
      }
      else if (lastDataPoint < dataPoint) {
        trendcount++;
      }
      else if (lastDataPoint > dataPoint) {
        trendcount = 0L;
      }
      lastDataPoint = dataPoint;
    }
  }

  if (this.hasViews()) {
    EventBean newDataPost = populateMap(trendcount);
    updateChildren(new EventBean[] {newDataPost}, new EventBean[] {oldDataPost});
  }
}
```

This `update` method must adhere to the following view conventions, to prevent memory leaks and to enable correct behavior within the engine:

- Views must post a remove stream in the form of old data to child views. The remove stream must consist of the same event reference(s) posted as insert stream (new data).

Please refer to the sample views for a code sample on how to implement `iterator` and `cloneView` methods.

8.2.3. Configuring View Namespace and Name

The view factory class name as well as the view namespace and name for the new view must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-view namespace="custom" name="trendspotter"
      factory-class="net.esper.regression.view.MyTrendSpotterViewFactory" />
</esper-configuration>
```

The new view is now ready to use in a statement:

```
select * from StockTick.custom:trendspotter('price')
```

Note that the view must implement the `copyView` method to enable the view to work in a grouping context as shown in the next statement:

```
select * from StockTick.std:groupby('symbol').custom:trendspotter('price')
```

8.3. Custom Aggregation Functions

Aggregation functions aggregate event property values or expression results obtained from one or more streams. Examples for built-in aggregation functions are `count(*)`, `sum(price * volume)` or `avg(distinct volume)`.

The optional keyword `distinct` ensures that only distinct (unique) values are aggregated and duplicate values are ignored by the aggregation function. Custom plug-in aggregation functions do not need to implement the logic to handle `distinct` values. This is because when the engine encounters the `distinct` keyword, it eliminates any non-distinct values before passing the value for aggregation to the custom aggregation function.

The following steps are required to develop and use a custom aggregation function with Esper.

1. Implement an aggregation function class.
2. Register the aggregation function class with the engine by supplying a function name, via the engine configuration file or the configuration API.

The code for the example aggregation function as shown in this chapter can be found in the test source folder in the package `net.esper.regression.client` by the name `MyConcatAggregationFunction`. The sample function simply concatenates string-type values.

8.3.1. Implementing an Aggregation Function

An aggregation function class is responsible for the following functions:

- Implement a `validate` method that validates the value type of the data points that the function must process.
- Implement a `getValueType` method that returns the type of the aggregation value generated by the function. For example, the built-in `count` aggregation function returns `Long.class` as it generates `long` -typed values.
- Implement an `enter` method that the engine invokes to add a data point into the aggregation, when an event enters a data window
- Implement a `leave` method that the engine invokes to remove a data point from the aggregation, when an event leaves a data window

- Implement a `getValue` method that returns the current value of the aggregation.

Aggregation function classes simply subclass `net.esper.eql.agg.AggregationSupport`:

```
public class MyConcatAggregationFunction extends AggregationSupport { ...
```

The engine generally constructs one instance of the aggregation function class for each time the function is listed in a statement, however the engine may decide to reduce the number of aggregation class instances if it finds equivalent aggregations. The constructor initializes the aggregation function:

```
public class MyConcatAggregationFunction extends AggregationSupport {
  private final static char DELIMITER = ' ';
  private StringBuilder builder;
  private String delimiter;

  public MyConcatAggregationFunction()
  {
    super();
    builder = new StringBuilder();
    delimiter = "";
  }
  ...
```

An aggregation function must provide an implementation of the `validate` method that is passed the result type of the expression within the aggregation function. Since the example concatenation function requires string types, it implements a type check:

```
public void validate(Class childNodeType) {
  if (childNodeType != String.class) {
    throw new IllegalArgumentException("Concat aggregation requires a String parameter");
  }
}
```

The `enter` method adds a datapoint to the current aggregation value. The example `enter` method shown below adds a delimiter and the string value to a string buffer:

```
public void enter(Object value) {
  if (value != null) {
    builder.append(delimiter);
    builder.append(value.toString());
    delimiter = String.valueOf(DELIMITER);
  }
}
```

Conversly, the `leave` method removes a datapoint from the current aggregation value. The example `leave` method removes from the string buffer:

```
public void leave(Object value) {
  if (value != null) {
    builder.delete(0, value.toString().length() + 1);
  }
}
```

In order for the engine to validate the type returned by the aggregation function against the types expected by enclosing expressions, the `getValueType` must return the result type of any values produced by the aggregation function:

```
public Class getValueType() {
  return String.class;
}
```

Finally, the engine obtains the current aggregation value by means of the `getValue` method:

```
public Object getValue() {
  return builder.toString();
}
```

8.3.2. Configuring Aggregation Function Name

The aggregation function class name as well as the function name for the new aggregation function must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-aggregation-function name="concat"
    function-class="net.esper.regression.client.MyConcatAggregationFunction" />
</esper-configuration>
```

The new aggregation function is now ready to use in a statement:

```
select concat(symbol) from StockTick.win:length(3)
```

8.4. Custom Pattern Guard

Pattern guards are pattern objects that control the lifecycle of the guarded sub-expression, and can filter the events fired by the subexpression.

The following steps are required to develop and use a custom guard object with Esper.

1. Implement a guard factory class, responsible for creating guard object instances.
2. Implement a guard class.
3. Register the guard factory class with the engine by supplying a namespace and name, via the engine configuration file or the configuration API.

The code for the example guard object as shown in this chapter can be found in the test source folder in the package `net.esper.regression.client` by the name `MyCountToPatternGuardFactory`. The sample guard discussed here counts the number of events occurring up to a maximum number of events, and end the subexpression when that maximum is reached.

8.4.1. Implementing a Guard Factory

A guard factory class is responsible for the following functions:

- Implement a `setGuardParameters` method that validates guard parameters.
- Implement a `makeGuard` method that constructs a new guard instance.

Guard factory classes subclass `net.esper.pattern.guard.GuardFactorySupport`:

```
public class MyCountToPatternGuardFactory extends GuardFactorySupport { ...
```

The engine constructs one instance of the guard factory class for each time the guard is listed in a statement.

The guard factory class implements the `setGuardParameters` method that is passed the parameters to the guard as supplied by the statement. It verifies the guard parameters, similar to the code snippet shown next. Our ex-

ample counter guard takes a single numeric parameter:

```
public void setGuardParameters(List<Object> guardParameters) throws GuardParameterException {
  if (guardParameters.size() != 1) {
    throw new GuardParameterException("Count-to guard takes a single integer parameter");
  }
  if (!(guardParameters.get(0) instanceof Integer)) {
    throw new GuardParameterException("Count-to guard takes a single integer parameter");
  }
  numCountTo = (Integer) guardParameters.get(0);
}
```

The `makeGuard` method is called by the engine to create a new guard instance. The example `makeGuard` method shown below passes the maximum count of events to the guard instance. It also passes a `Quitable` implementation to the guard instance. The guard uses `Quitable` to indicate that the sub-expression contained within must stop (quit) listening for events.

```
public Guard makeGuard(PatternContext context, Quitable quitable,
  Object stateNodeId, Object guardState) {
    return new MyCountToPatternGuard(numCountTo, quitable);
}
```

8.4.2. Implementing a Guard Class

A guard class has the following responsibilities:

- Provides a `startGuard` method that initalizes the guard.
- Provides a `stopGuard` method that stops the guard, called by the engine when the whole pattern is stopped, or the sub-expression containing the guard is stopped.
- Provides an `inspect` method that the pattern engine invokes to determine if the guard lets matching events pass for further evaluation by the containing expression.

Guard classes subclass `net.esper.pattern.guard.GuardSupport` as shown here:

```
public abstract class GuardSupport implements Guard { ...
```

The engine invokes the guard factory class to construct an instance of the guard class for each new sub-expression instance within a statement.

A guard class must provide an implementation of the `startGuard` method that the pattern engine invokes to start a guard instance. In our example, the method resets the guard's counter to zero:

```
public void startGuard() {
  counter = 0;
}
```

The pattern engine invokes the `inspect` method for each time the sub-expression indicates a new event result. Our example guard needs to count the number of events matched, and quit if the maximum number is reached:

```
public boolean inspect(MatchedEventMap matchEvent) {
  counter++;
  if (counter > numCountTo) {
    quitable.guardQuit();
    return false;
  }
  return true;
}
```

The `inspect` method returns true for events that pass the guard, and false for events that should not pass the

guard.

8.4.3. Configuring Guard Namespace and Name

The guard factory class name as well as the namespace and name for the new guard must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-pattern-guard namespace="myplugin" name="count_to"
      factory-class="net.esper.regression.client.MyCountToPatternGuardFactory"/>
</esper-configuration>
```

The new guard is now ready to use in a statement. The next pattern statement detects the first 10 MyEvent events:

```
select * from pattern [(every MyEvent) where myplugin:count_to(10)]
```

Note that the `every` keyword was placed within parentheses to ensure the guard controls the repeated matching of events.

8.5. Custom Pattern Observer

Pattern observers are pattern objects that are executed as part of a pattern expression and can observe events or test conditions. Examples for built-in observers are `timer:at` and `timer:interval`. Some suggested uses of observer objects are:

- Implement custom scheduling logic using the engine's own scheduling and timer services
- Test conditions related to prior events matching an expression

The following steps are required to develop and use a custom observer object within pattern statements:

1. Implement an observer factory class, responsible for creating observer object instances.
2. Implement an observer class.
3. Register an observer factory class with the engine by supplying a namespace and name, via the engine configuration file or the configuration API.

The code for the example observer object as shown in this chapter can be found in the test source folder in package `net.esper.regression.client` by the name `MyFileExistsObserver`. The sample observer discussed here very simply checks if a file exists, using the filename supplied by the pattern statement, and via the `java.io.File` class.

8.5.1. Implementing an Observer Factory

An observer factory class is responsible for the following functions:

- Implement a `setObserverParameters` method that validates observer parameters.
- Implement a `makeObserver` method that constructs a new observer instance.

Observer factory classes subclass `net.esper.pattern.observer.ObserverFactorySupport`:

```
public class MyFileExistsObserverFactory extends ObserverFactorySupport { ...
```

The engine constructs one instance of the observer factory class for each time the observer is listed in a statement.

The observer factory class implements the `setObserverParameters` method that is passed the parameters to the observer as supplied by the statement. It verifies the observer parameters, similar to the code snippet shown next. Our example file-exists observer takes a single string parameter:

```
public void setObserverParameters(List<Object> observerParameters)
                        throws ObserverParameterException {
  String message = "File exists observer takes a single string filename parameter";
  if (observerParameters.size() != 1) {
    throw new ObserverParameterException(message);
  }
  if (!(observerParameters.get(0) instanceof String)) {
    throw new ObserverParameterException(message);
  }

  filename = observerParameters.get(0).toString();
}
```

The pattern engine calls the `makeObserver` method to create a new observer instance. The example `makeObserver` method shown below passes parameters to the observer instance:

```
public EventObserver makeObserver(PatternContext context,
                        MatchedEventMap beginState,
                        ObserverEventEvaluator observerEventEvaluator,
                        Object stateNodeId,
                        Object observerState) {
  return new MyFileExistsObserver(beginState, observerEventEvaluator, filename);
}
```

The `ObserverEventEvaluator` parameter allows an observer to indicate events, and to indicate change of truth value to permanently false. Use this interface to indicate when your observer has received or witnessed an event, or changed it's truth value to true or permanently false.

The `MatchedEventMap` parameter provides a Map of all matching events for the expression prior to the observer's start. For example, consider a pattern as below:

```
a=MyEvent -> myplugin:my_observer(...)
```

The above pattern tagged the MyEvent instance with the tag "a". The pattern engine starts an instance of `my_observer` when it receives the first MyEvent. The observer can query the `MatchedEventMap` using "a" as a key and obtain the tagged event.

8.5.2. Implementing an Observer Class

An observer class has the following responsibilities:

- Provides a `startObserve` method that starts the observer.
- Provides a `stopObserve` method that stops the observer, called by the engine when the whole pattern is stopped, or the sub-expression containing the observer is stopped.

Observer classes subclass `net.esper.pattern.observer.ObserverSupport` as shown here:

```
public class MyFileExistsObserver implements EventObserver { ...
```

The engine invokes the observer factory class to construct an instance of the observer class for each new sub-

expression instance within a statement.

An observer class must provide an implementation of the `startObserve` method that the pattern engine invokes to start an observer instance. In our example, the observer checks for the presence of a file and indicates the truth value to the remainder of the expression:

```
public void startObserve() {
  File file = new File(filename);
  if (file.exists()) {
    observerEventEvaluator.observerEvaluateTrue(beginState);
  }
  else {
    observerEventEvaluator.observerEvaluateFalse();
  }
}
```

Note the observer passes the `ObserverEventEvaluator` an instance of `MatchedEventMap`. The observer can also create one or more new events and pass these events through the Map to the remaining expressions in the pattern.

8.5.3. Configuring Observer Namespace and Name

The observer factory class name as well as the namespace and name for the new observer must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-pattern-observer namespace="myplugin" name="file_exists"
    factory-class="net.esper.regression.client.MyFileExistsObserverFactory" />
</esper-configuration>
```

The new observer is now ready to use in a statement. The next pattern statement checks every 10 seconds if the given file exists, and indicates to the listener when the file is found.

```
select * from pattern [every timer:interval(10 sec) -> myplugin:file_exists("myfile.txt")]
```

Chapter 9. Examples, Tutorials, Case Studies

The tutorial and case studies are available on the public web site at `http://esper.codehaus.org/evaluating/evaluating.html`.

9.1. Examples Overview

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

In order to compile and run the samples please follow the below instructions:

1. Make sure Java 1.5 or greater is installed and the JAVA_HOME environment variable is set.
2. Open a console window and change directory to examples/etc.
3. Run "setenv.bat" (Windows) or "setenv.sh" (Unix) to verify your environment settings.
4. Run "compile.bat" (Windows) or "compile.sh" (Unix) to compile the examples.
5. Now you are ready to run the examples. Some examples require mandatory parameters. Further information to running each example can be found in the "examples/etc" folder in file "readme.txt".
6. Modify the logger logging level in the "log4j.xml" configuration file changing DEBUG to INFO on a class or package level to reduce the volume of text output.

JUnit tests exist for the example code. The JUnit test source code for the examples can be found in the `examples/test` folder. To build and run the example 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. Market Data Feed Monitor

This example processes a raw market data feed. It reports throughput statistics and detects when the data rate of a feed falls off unexpectedly. A rate fall-off may mean that the data is stale and we want to alert when there is a possible problem with the feed.

The classes for this example live in package `net.esper.example.marketdatafeed`. Run "run_mktdatafeed.bat" (Windows) or "run_mktdatafeed.sh" (Unix) in the `examples/etc` folder to start the market data feed simulator.

9.2.1. Input Events

The input stream consists of 1 event stream that contains 2 simulated market data feeds. Each individual event in the stream indicates the feed that supplies the market data, the security symbol and some pricing information:

```
String symbol;
FeedEnum feed;
double bidPrice;
double askPrice;
```

9.2.2. Computing Rates Per Feed

For the throughput statistics and to detect rapid fall-off we calculate a ticks per second rate for each market data feed.

We can use an EQL statement that specifies a view onto the market data event stream that batches together 1 second of events. We specify the feed and a count of events per feed as output values. To make this data available for further processing, we insert output events into the TicksPerSecond event stream:

```
insert into TicksPerSecond
select feed, count(*) as cnt
  from MarketDataEvent.win:time_batch(1 second)
 group by feed
```

9.2.3. Detecting a Fall-off

We define a rapid fall-off by alerting when the number of ticks per second for any second falls below 75% of the average number of ticks per second over the last 10 seconds.

We can compute the average number of ticks per second over the last 10 seconds simply by using the TicksPerSecond events computed by the prior statement and averaging the last 10 seconds. Next, we compare the current rate with the moving average and filter out any rates that fall below 75% of the average:

```
select feed, avg(cnt) as avgCnt, cnt as feedCnt
  from TicksPerSecond.win:time(10 seconds)
 group by feed
having cnt < avg(cnt) * 0.75
```

9.2.4. Event generator

The simulator generates market data events for 2 feeds, feed A and feed B. The first parameter to the simulator is a number of threads. Each thread sends events for each feed in an endless loop. Note that as the Java VM garbage collection kicks in, the example generates rate drop-offs during such pauses.

The second parameter is a rate drop probability parameter specifies the probability in percent that the simulator drops the rate for a randomly chosen feed to 60% of the target rate for that second. Thus rate fall-off alerts can be generated.

The third parameter defines the number of seconds to run the example.

9.3. Transaction 3-Event Challenge

The classes for this example live in package `net.esper.example.transaction`. Run "run_txnsim.bat" (Windows) or "run_txnsim.sh" (Unix) to start the transaction simulator. Please see the readme file in the same folder for build instructions and command line parameters.

9.3.1. The Events

The use case involves tracking three components of a transaction. It‘s important that we use at least three components, since some engines have different performance or coding for only two events per transaction. Each component comes to the engine as an event with the following fields:

- Transaction ID
- Time stamp

In addition, we have the following extra fields:

In event A:

- Customer ID

In event C:

- Supplier ID (the ID of the supplier that the order was filled through)

9.3.2. Combined event

We need to take in events A, B and C and produce a single, combined event with the following fields:

- Transaction ID
- Customer ID
- Time stamp from event A
- Time stamp from event B
- Time stamp from event C

What we‘re doing here is matching the transaction IDs on each event, to form an aggregate event. If all these events were in a relational database, this could be done as a simple SQL join… except that with 10,000 events per second, you will need some serious database hardware to do it.

9.3.3. Real time summary data

Further, we need to produce the following:

- Min,Max,Average total latency from the events (difference in time between A and C) over the past 30 minutes.
- Min,Max,Average latency grouped by (a) customer ID and (b) supplier ID. In other words, metrics on the the latency of the orders coming from each customer and going to each supplier.
- Min,Max,Average latency between events A/B (time stamp of B minus A) and B/C (time stamp of C minus B).

9.3.4. Find problems

We need to detect a transaction that did not make it through all three events. In other words, a transaction with events A or B, but not C. Note that, in this case, what we care about is event C. The lack of events A or B could indicate a failure in the event transport and should be ignored. Although the lack of an event C could also be a transport failure, it merits looking into.

9.3.5. Event generator

To make testing easier, standard and to demonstrate how the example works, the example is including an event generator. The generator generates events for a given number of transactions, using the following rules:

- One in 5,000 transactions will skip event A
- One in 1,000 transactions will skip event B
- One in 10,000 transactions will skip event C.
- Transaction identifiers are randomly generated
- Customer and supplier identifiers are randomly chosen from two lists

- The time stamp on each event is based on the system time. Between events A and B as well as B and C, between 0 and 999 is added to the time. So, we have an expected time difference of around 500 milliseconds between each event
- Events are randomly shuffled as described below

To make things harder, we don‘t want transaction events coming in order. This code ensures that they come completely out of order. To do this, we fill in a bucket with events and, when the bucket is full, we shuffle it. The buckets are sized so that some transactions‘ events will be split between buckets. So, you have a fairly randomized flow of events, representing the worst case from a big, distributed infrastructure.

The generator lets you change the size of the bucket (small, medium, large, larger, largerer). The larger the bucket size, the more events potentially come in between two events in a given transaction and so, the more the performance characteristics like buffers, hashes/indexes and other structures are put to the test as the bucket size increases.

9.4. J2EE Self-Service Terminal Management

The example is about a J2EE-based self-service terminal managing system in an airport that gets a lot of events from connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as 'paper low' or 'terminal out of order'. Other events observe activity as customers use a terminal to check in and print boarding tickets.

9.4.1. Events

Each self-service terminal can publish any of the 6 events below.

- Checkin - Indicates a customer started a check-in dialog
- Cancelled - Indicates a customer cancelled a check-in dialog
- Completed - Indicates a customer completed a check-in dialog
- OutOfOrder - Indicates the terminal detected a hardware problem
- LowPaper - Indicates the terminal is low on paper
- Status - Indicates terminal status, published every 1 minute regardless of activity as a terminal heartbeat

All events provide information about the terminal that published the event, and a timestamp. The terminal information is held in a property named "term" and provides a terminal id. Since all events carry similar information, we model each event as a subtype to a base class BaseTerminalEvent, which will provide the terminal information that all events share. This enables us to treat all terminal events polymorphically, that is we can treat derived event types just like their parent event types. This helps simplify our queries.

All terminals publish Status events every 1 minute. In normal cases, the Status events indicate that a terminal is alive and online. The absence of status events may indicate that a terminal went offline for some reason and that may need to be investigated.

9.4.2. Detecting Customer Check-in Issues

A customer may be in the middle of a check-in when the terminal detects a hardware problem or when the network goes down. In that situation we want to alert a team member to help the customer. When the terminal detects a problem, it issues an OutOfOrder event. A pattern can find situations where the terminal indicates out-of-order and the customer is in the middle of the check-in process:

```
select * from pattern [ every a=Checkin ->
      ( OutOfOrder(term.id=a.term.id) and not
          (Cancelled(term.id=a.term.id) or Completed(term.id=a.term.id)) )]
```

9.4.3. Absence of Status Events

Since Status events arrive in regular intervals of 60 seconds, we can make us of temporal pattern matching using timer to find events that didn't arrive. We can use the every operator and timer:interval() to repeat an action every 60 seconds. Then we combine this with a not operator to check for absence of Status events. A 65 second interval during which we look for Status events allows 5 seconds to account for a possible delay in transmission or processing:

```
select 'terminal 1 is offline' from pattern
  [every timer:interval(60 sec) -> (timer:interval(65 sec) and not Status(term.id = 'T1'))]
output first every 5 minutes
```

9.4.4. Activity Summary Data

By presenting statistical information about terminal activity to our staff in real-time we enable them to monitor the system and spot problems. The next example query simply gives us a count per event type every 1 minute. We could further use this data, available through the CountPerType event stream, to join and compare against a recorded usage pattern, or to just summarize activity in real-time.

```
insert into CountPerType
select type, count(*) as countPerType
from BaseTerminalEvent.win:time(10 minutes)
group by type
output all every 1 minutes
```

9.4.5. Sample Application for J2EE Application Server

The example code in the distribution package implements a message-driven enterprise java bean (MDB EJB). We used an MDB as a convenient place for processing incoming events via a JMS message queue or topic. The example uses 2 JMS queues: One queue to receive events published by terminals, and a second queue to indicate situations detected via EQL statement and listener back to a receiving process.

This example has been packaged for deployment into a JBoss Java application server (see http://www.jboss.org) with default deployment configuration. JBoss is an open-source application server available under LGPL license. Of course the choice of application server does not indicate a requirement or preference for the use of Esper in a J2EE container. Other quality J2EE application servers are available and perhaps more suitable to run this example or a similar application.

The complete example code can be found in the "examples/terminalsvc" folder of the distribution. The Java package name is net.esper.example.terminalsvc.

Running the Example

The pre-build EAR file contains the MDB for deployment to a JBoss application server with default deployment options. The JBoss default configuration provides 2 queues that this example utilizes: queue/A and queue/B. The queue/B is used to send events into the MDB, while queue/A is used to indicate back the any data received by listeners to EQL statements.

The application can be deployed by copying the ear file in the "examples/terminalsvc/terminalsvc-ear" folder to your JBoss deployment directory located under the JBoss home directory under "server/default/deploy".

The example contains an event simulator and an event receiver that can be invoked from the command line. See the folder "examples/terminalsvc/etc" folder readme file and start scripts for Windows and Unix, and the docu-

mentation set for further information on the simulator.

Building the Example

This example requires Maven 2 to build. To build the example, change directory to the folder "examples/terminalsvc" and type "mvn package". The instructions have been tested with JBoss AS 4.0.4.GA and Maven 2.0.4.

The Maven build packages the EAR file for deployment to a JBoss application server with default deployment options.

Running the Event Simulator and Receiver

The example also contains an event simulator that generates meaningful events. The simulator can be run from the directory "examples/terminalsvc/etc" via the command "run_terminalsvc_sender.bat" (Windows) and "run_terminalsvc_sender.sh" (Linux). The event simulator generates a batch of at least 200 events every 1 second. Randomly, with a chance of 1 in 10 for each batch of events, the simulator generates either an OutOfOrder or a LowPaper event for a random terminal. Each batch the simulator generates 100 random terminal ids and generates a Checkin event for each. It then generates either a Cancelled or a Completed event for each. With a chance of 1 in 1000, it generates an OutOfOrder event instead of the Cancelled or Completed event for a terminal.

The event receiver listens to the MDB-outcoming queue for alerts and prints these out to console. The receiver can be run from the directory "examples/terminalsvc/etc" via the command "run_terminalsvc_receiver.bat" (Windows) and "run_terminalsvc_receiver.sh" (Linux).

9.5. Assets Moving Across Zones - An RFID Example

This example out of the RFID domain processes location report events. Each location report event indicates an asset id and the current zone of the asset. The example solves the problem that when a given set of assets is not moving together from zone to zone, then an alert must be fired.

Each asset group is tracked by 2 statements. The two statements to track a single asset group consisting of assets identified by asset ids {1, 2, 3} are as follows:

```
insert into CountZone_G1
select 1 as groupId, zone, count(*) as cnt
from LocationReport(assetId in 1, 2, 3).std:unique('assetId')
group by zone

select Part.zone from pattern [
  every Part=CountZone_G1(cnt in (1,2)) ->
    (timer:interval(10 sec)  and not CountZone_G1(zone=Part.zone, cnt in (0,3)))]
```

The classes for this example can be found in package `net.esper.example.rfid`.

This example provides a Swing-based GUI that can be run from the command line. The GUI allows drag-and-drop of three RFID tags that form one asset group from zone to zone. Each time you move an asset across the screen the example sends an event into the engine indicating the asset id and current zone. The example detects if within 10 seconds the three assets do not join each other within the same zone, but stay split across zones. Run "run_rfid_swing.bat" (Windows) or "run_rfid_swing.sh" (Unix) to start the example's Swing GUI.

The example also provides a simulator that can be run from the command line. The simulator generates a number of asset groups as specified by a command line argument and starts a number of threads as specified by a

command line argument to send location report events into the engine. Run "run_rfid_sim.bat" (Windows) or "run_rfid_sim.sh" (Unix) to start the RFID location report event simulator. Please see the readme file in the same folder for build instructions and command line parameters.

9.6. AutoID RFID Reader generating XML documents

In this example an array of RFID readers sense RFID tags as pallets are coming within the range of one of the readers. A reader generates XML documents with observation information such as reader sensor ID, observation time and tags observed. A statement computes the total number of tags per reader sensor ID within the last 60 seconds.

This example demonstrates how XML documents unmarshalled to `org.w3c.dom.Node` DOM document nodes can natively be processed by the engine without requiring Java object event representations. The example uses an XPath expression for an event property counting the number of tags observed by a sensor. The XML documents follow the AutoID (`http://www.autoid.org/`) organization standard.

The classes for this example can be found in package `net.esper.example.autoid`. As events are XML documents with no Java object representation, the example does not have event classes.

A simulator that can be run from the command line is also available for this example. The simulator generates a number of XML documents as specified by a command line argument and prints out the totals per sensor. Run "run_autoid.bat" (Windows) or "run_autoid.sh" (Unix) to start the autoid simulator. Please see the readme file in the same folder for build instructions and command line parameters.

The code snippet below shows the simple statement to compute the total number of tags per sensor. The statement is created by class `net.esper.example.autoid.RFIDTagsPerSensorStmt`.

```
select ID as sensorId, sum(countTags) as numTagsPerSensor
from AutoIdRFIDExample.win:time(60 seconds)
where Observation[0].Command = 'READ_PALLET_TAGS_ONLY'
group by ID
```

9.7. 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

9.8. 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

9.9. QualityOfService

This example develops some code for measuring quality-of-service levels such as for a service-level agreement (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 its 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

9.10. 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 `examples/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.

9.11. StockTick RSI

The RSI gives you the trend for a stock and for more complete explanation, you can visit the link: http://www.stockcharts.com/education/IndicatorAnalysis/indic_RSI.html.

After a definite number of stock events, or accumulation period, the first RSI is computed. Then for each subsequent stock event, the RSI calculations use the previous period's Average Gain and Loss to determine the "smoothed RSI".

Summary:

- Uses a simple event pattern with a filter which feeds a listener that computes the RSI, which publishes events containing the computed RSI.

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*.